R 软件开发自学视频教程



配套自学视频上百集
编程实例分析上千个
典型模块分析近20类
项目案例分析近20套
《程序员求职面试宝典》电子书1部

C++ 自学视频教程

79集配套视频 + 2100项配套资源

- 编程实例分析961个
- 典型模块分析15类
- 项目案例分析16套
- 数学及逻辑思维、IT英语能力测试311项
- 实践训练任务616项
- 《程序员求职面试宝典》（面试真题、技巧及职业规划）1部

软件开发技术联盟◎编著

◎配套自学视频上百集

几乎覆盖全书所有实例，先听视频讲解，再仿照书中实例实践，会大幅提高学习效率。

◎编程实例分析上千个

各类实例一应俱全，无论学习哪一章节，都可从中找到相关实例加以练习，相信对深入学习极有帮助。

◎典型模块分析近20类

既可作为综合应用实例学习，又可将模块移植到相关应用中，从而避免重复劳动，提高工作效率。

◎项目案例分析近20套

从需求分析、系统设计、模块分解到代码实现，几乎展现了项目开发的整个过程。

◎《程序员求职面试宝典》电子书1部

各类面试真题、面试技巧、程序员职业生涯、简历设计、IT企业中的自身修养等帮助读者更好就业和长远发展。

清华大学出版社

软件开发自学视频教程

C++自学视频教程

软件开发技术联盟　编著

清华大学出版社

北　京

内 容 简 介

《C++自学视频教程》以初学者为主要对象，全面介绍了C++程序设计相关的各种技术。在内容排列上由浅入深，让读者循序渐进地掌握C++编程技术；在内容讲解上结合丰富的图解和形象的比喻，帮助读者理解晦涩难懂的技术；在内容形式上附有大量的注意、说明、技巧等栏目，夯实读者理论技术，丰富管理与开发经验。

《C++自学视频教程》共分3篇18章，其中，第1篇为入门篇，主要包括初识C++、认识C++程序、变量和数据类型、运算符与表达式、条件判断语句、循环控制语句、封装函数使程序模块化、C++中的指针、C++中的引用、使用数组获取连续空间等内容；第2篇为提高篇，主要包括面向对象编程、从基类到派生类、C++模板的使用、代码整理、掌握C++标准模板库、利用文件处理数据等内容；第3篇为实战篇，主要包括ATM机界面、猜数字游戏、吃豆子游戏和人事考勤管理系统4个实战项目。另外本书光盘含：

17小时视频讲解/961个编程实例/15个经典模块分析/16个项目开发案例/311个编程实践任务/616个能力测试题目（基础能力测试、数学及逻辑思维能力测试、面试能力测试、编程英语能力测试）/23个IT励志故事。

本书适用于C++编程的爱好者、初学者和中级开发人员，也可作为大中专院校和培训机构的教材。

图书在版编目（CIP）数据

C++自学视频教程/软件开发技术联盟编著．—北京：清华大学出版社，2014
软件开发自学视频教程
ISBN 978-7-302-37102-1

I. ①C… II. ①软… III. ①C语言-程序设计-教材 IV. ①TP312

中国版本图书馆CIP数据核字（2014）第146019号

责任编辑：赵洛育
封面设计：李志伟
版式设计：文森时代
责任校对：王　云
责任印制：

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦A座　　**邮　　编：**100084
社 总 机：010-62770175　　**邮　　购：**010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　　销：全国新华书店
开　　本：203mm×260mm　**印　　张：**33　**字　　数：**862千字
（附DVD1张）
版　　次：2014年12月第1版　**印　　次：**2014年12月第1次印刷
印　　数：1～4000
定　　价：79.80元

产品编号：051612-01

前　言

Preface

本书编写背景

为什么一方面很多毕业生不太容易找到工作，另一方面很多企业却招不到合适的人才？为什么很多学生学习很刻苦，临毕业了却感到自己似乎什么都不会？为什么很多学生到企业之后，发现很多所学的知识用不上？……高校课程设置与企业应用严重脱节，高校所学知识得不到很好的实践，本来是为了实际应用而学习却变成了应付考试，是造成如上所述现象的主要原因。

为了能满足社会需要，有些人不得不花费巨额费用、花费半年到一年时间到社会再培训，浪费了巨大的人力物力。有没有一种办法让学生在校就能学到企业应用的内容呢？——本书就是为此目的而来。本书从没有编程基础或稍有编程基础的读者层次开始，通过适合自学的方式，从基础知识到小型实例到综合实例到项目案例，让学生在学校就能学到企业应用的内容，从而实现从学校所学到企业应用的重大跨越，架起从学校通向社会的桥梁。

本书特点

1．从基础到项目实战，快速铺就就业之路

全书体例为：基础知识+小型实例+综合实例+项目实战，既符合循序渐进的学习规律，也力求贴近项目实战等实际应用。基础知识是必备内容；小型实例则通过实例巩固基础知识；综合实例则是在进一步综合应用基础知识的前提下，通过模块的形式让内容更加贴近实际应用；项目实战则是展现项目开发的全过程，让读者对基本的项目开发有一个全面的认识。

2．全程配套视频讲解，让老师手把手教您

本书配书光盘含配套视频讲解，基本覆盖全书内容，学习之前，先看、听视频讲解，然后对照书模仿练习，相信会快速提高学习效率。

3．配套资源极为丰富，各类实例一应俱全

（1）**实例资源库**：包括**上千个编程实例，各种类型一应俱全**，无论学习这本书的哪一章节，都可以从中找到相关的多种实例加以实践，相信对深入学习极有帮助。

（2）**模块资源库**：包括了最常用的**十多个经典模块分析**，它们既可作为**综合应用实例学习，又可移植到相关应用中**，进而避免重复劳动，提高工作效率。

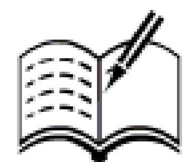
（3）**项目（案例）资源库**：包括**十多个项目开发案例，从需求分析、系统设计、模块分析到代码实现**，几乎全程展现了项目开发的整个过程。

（4）**任务（训练）资源库**：共计**千余个实践任务**，读者可以自行实践练习，还可以到对应的网站上寻找答案。

（5）**能力测试资源库**：列举了**几百个能力测试题目，包括编程基础能力测试、数学及逻辑思维能力测试、面试能力测试、编程英语能力测试**，便于读者自我测试。

（6）**编程人生**：精选了**二十多个 IT 励志故事**，希望读者朋友从这些 IT 成功人士的经历中**汲取精神力量**，让这些经历成为您不断进取、勇攀高峰的强大精神动力。





Note

如何高效使用本书

建议首先看相关实例视频，然后对照图书的实例，动手操作或者运行程序，反复体会，之后再打开本书光盘的“自主学习系统”，找一些对应的实例练习。当然，还可以参考“自主学习系统”的其他资源，加以补充和拓展。

本书常见问题

1．编程软件的获取

按照本书上的实例进行操作练习，需要事先在电脑上安装相关的语言或工具的开发环境（编程软件）。本书光盘只提供了教学视频、自主学习系统等辅助资料，并未提供编程软件，读者朋友需要在网上搜索下载，或者到当地电脑城、软件经销商处购买。

2．关于本书的技术问题或有关本书信息的发布

（1）读者朋友遇到有关本书的技术问题，建议先登录：www.rjkflm.com，搜索到本书后，查看该书的留言是否已经对您的相关问题进行了回复，以避免浪费您更多的时间。

（2）如果留言没有相关问题，可加入 QQ：4006751066 咨询有关本书的技术问题。

（3）本书经过多次审校，仍然可能有极少数错误，欢迎读者朋友批评指正，请给我们留言，我们也将对提出问题和建议的读者予以奖励。另外，有关本书的勘误，我们会在 www.rjkflm.com 网站上公布。

3．关于本书光盘的使用

本书光盘只能在电脑光驱（DVD 格式）中使用，光盘中的视频文件双击即可自行播放。极个别光盘视频文件如果不能打开，请暂时关闭一下杀毒软件再打开；若仍然无法打开，建议换台电脑后将光盘内容复制过来后打开（极个别光驱与光盘不兼容导致无法读取的现象是有的）。另外，盘面若有胶水等脏物建议先行擦拭干净。

关于作者

本书由软件开发技术联盟组织编写。该联盟由一家有十多年集软件开发、数字教育、图书出版为一体的高科技公司——明日科技和一些中青年骨干教师组成。

本书主要由崔佳音、周佳星执笔编写，其他参与本书编写的人员有王小科、王国辉、张鑫、杨丽、陈英、高春艳、刘莉莉、赛奎春、刘佳、辛洪郁、郭铁、张金辉、高润岭、王敬杰、高茹、任媛、孙桂杰、李贺、陈威、高飞、刘志铭、宋晶、宋禹蒙、王雨竹、张彦国、张磊、刘丽艳、邹淑芳、王喜平、于国槐、郭锐、刘红艳、郭鑫、李根福等。

寄语读者

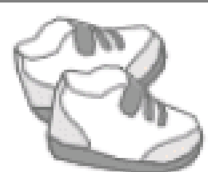
亲爱的读者朋友，千里有缘一线牵，感谢您在茫茫书海中找到了本书，希望她架起你我之间学习、友谊的桥梁，希望她带您轻松步入妙趣横生的编程世界，希望她成为您成长道路上的铺路石。

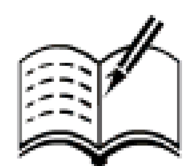
软件开发技术联盟

目　录

Contents

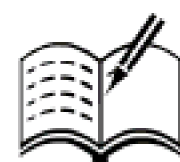
第1篇　入　门　篇

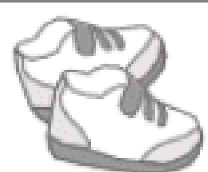


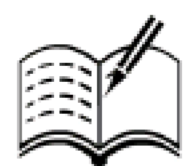


Note



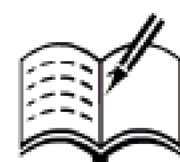

Note



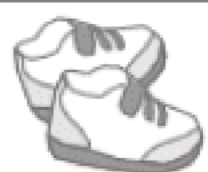


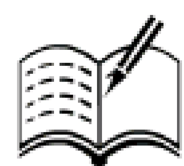
第 2 篇 提 高 篇





Note



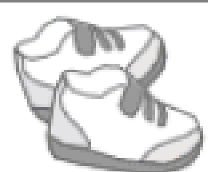


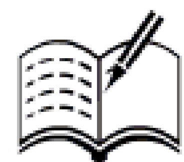
Note





第 3 篇 实 战 篇




Note

本书光盘“自主学习系统”（各类学习资源库）

内容索引

说明：

亲爱的读者朋友，熟练掌握一门编程工具，一本书是远远不够的。为了方便您深入学习、拓展视野，我们开发整理了海量的学习资源库，即配书光盘中的“自主学习系统”，内容有6大部分：

1. **实例资源库：**包括**961个编程实例，各种类型一应俱全**，无论学习这本书的哪一章节，都可以从中找到相关的多种实例加以实践，相信对深入学习极有帮助。

2. **模块资源库：**包括了最常用的**15个经典模块分析**，它们既可作为**综合应用实例学习，又可移植到相关应用中**，进而避免重复劳动，提高工作效率。

3. **项目（案例）资源库：**包括**16个项目开发案例，从需求分析、系统设计、模块分析到代码实现**，几乎全程展现了项目开发的整个过程。

4. **任务（训练）资源库：**共计**311个编程实践任务**，读者可以自行实践练习，还可以到对应的网站上寻找答案。

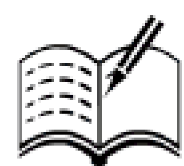
5. **能力测试资源库：**列举了**616道能力测试题目，包括编程基础能力测试、数学及逻辑思维能力测试、面试能力测试、编程英语能力测试**，便于读者自我测试。

6. **编程人生：**精选了**23个IT励志故事**，希望读者朋友从这些IT成功人士的经历中**汲取精神力量**，让这些经历成为您不断进取、勇攀高峰的强大精神动力。

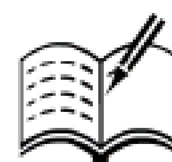
第1部分　实例资源库

（961个完整实例分析）









Note

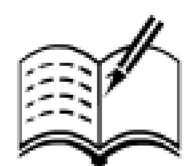


Note




Note

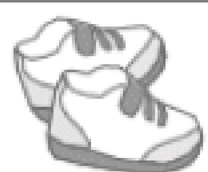



Note





Note

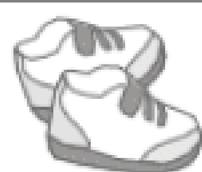


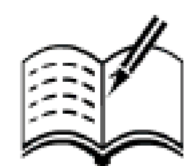


Note




Note

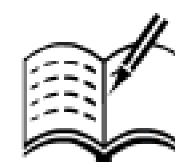




Note

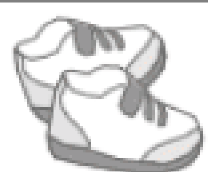
第 2 部分　模块资源库
（15 个经典模块分析）





第 3 部分　项目资源库

（16 个项目开发案例）





Note

第 4 部分　任务资源库
（311 个编程实践任务）




Note





Note





第 5 部分　能力测试资源库

（616 道能力测试题目）

第 6 部分　编程人生

（23 个 IT 励志故事）

第 1 篇



入门篇

第1章

初识 C++

（视频讲解：45 分钟）

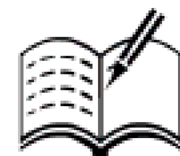
C++是当今流行的编程语言，它是在 C 语言基础上发展起来的，随着面向对象编程思想的发展，C++也融入了新的编程理念，这些理念有利于程序的开发。C++从语言角度说也是个规范，随着 C++11 标准的发布，部分编译器开始了支持新特性的先例。

本章能够完成的主要范例（已掌握的在方框中打勾）

- [ ] 了解 C++的发展历程
- [ ] 了解为 C++发展做出杰出贡献的人物
- [ ] 掌握主要的 C++编译器及开发环境
- [ ] 掌握 C++项目文件及编译工程
- [ ] 掌握 Visual Studio 2010 开发工具的安装和卸载
- [ ] 使用 Visual Studio 2010 创建 C++控制台应用程序



1.1　C++的历史背景



Note

学习一门语言，首先要对这门语言有一定的了解，知道它能做什么，要怎样做才能学好。本节将对 C++语言的历史背景进行简单的介绍，使读者对 C++语言有一个简单而直观的印象。

1.1.1　计算机的问世

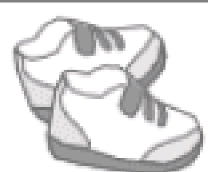
计算机的出现给我们的生活带来了巨大的变化，它是如何发展起来的呢？开始时人们致力于能够进行四则运算的机器，是通过机械齿轮运作的加法器，而后是精度只有 12 位的乘法计算器，直到 1847 年 Charles Babbages 开发出能计算 31 位精度的机械式差分机，这台差分机被普遍认为是世界第一台机械式计算机。随着电子物理的发展，真空二极管、真空三极管问世，到 1939 年第一部用真空管计算的机器被研制出来，该机器是能进行 16 位加法的机器；随后，氖气灯（霓虹灯）存储器、复杂数字计算机（断电器计数机）、可编写程序的计数机被一一研制出来。1946 年，第一台电子管计算机 ENIAC 在美国的宾夕法尼亚大学被研制出来，这台计算机占地 170 平方米，重 30 吨，有 1.8 万个电子管，用十进制计算，每秒运算 5000 次。计算机从此进入了电子计算机时代，经历了真空管计算机、晶体管计算机、集成电路计算机、大规模集成电路计算机 4 个阶段，每一个阶段都是随着电子物理的发展而发展的，晶体管的出现取代了电子管，将电子元件结合到一片小小的硅片上，形成集成电路（IC），在一个芯片上容纳几百个甚至几千个电子元件形成了大规模集成电路（LSI），直到现在已经出现了 32 纳米制作的电子芯片，可谓是发展迅速。计算机运行速度越来越快，从第一台计算机的每秒 5000 次到现在的 2GHz。

现在计算机已经应用到各个领域，科学计算、信号检测、数据管理、辅助设计都在使用计算机，人们的生活已经渐渐离不开它，所以说计算机是 20 世纪最伟大的发明。

1.1.2　C++发展历程

早期的计算机程序语言就是计算机控制指令，每条指令就是一组二进制数，不同的计算都有不同的计算机指令集。使用二进制指令集开发程序是件很头痛的事，需要记住大量的二进制数，为了便于记忆，人们将二进制数用字母组合代替，以字符串关键字代替二进制机器码的编程语言称为汇编语言，汇编语言被称为是低级语言，虽然它比机器码容易记忆，但仍然具有可读性差的缺点，大量的跳转指令和地址值很难让程序员在很短的时间理解程序的意思，于是编程语言进入了高级语言时代。

第一个高级语言是美国尤尼法克公司在 1952 年研制成功的 Short Code，但被广泛使用的高级语言是 FORTRAN，它是由美国科学家巴克斯设计并在 IBM 公司的计算机上实现的，但 FORTRAN 语言和 ALGOL60 主要应用于科学和工程计算，随后出现了 Pascal 和 C 语言。C 语言是在其他语言基础上发展起来的。首先是 Richard Martin 开发一种高级语言 BCPL，随后 Ken





Thompson 使用 BCPL 语言对其进行了简化，形成一门新的语言——B 语言，但 B 语言没有类型的概念，Dennis Ritchie 对 B 语言进行研究和改进，在 B 语言基础上添加了结构和类型，并将这个改进后的语言命名为 C 语言，寓意很简单，因为字母 C 是字母 B 的下一个字母，预示着语言的发展。

本书所讲述的 C++语言就是从 C 语言发展过来的，Stroustrup 经过钻研在 C 语言中加入类的概念，C++最初的名字是 C with Class，到 1983 年 12 月由 Rick Mascitti 建议改名为 CPlusPlus，即 C++。最开始提出类概念的语言是 Simula，它具有很高的灵活性，但无法胜任比较大型的程序，此后在 Simula 语言基础上发展的语言 Smalltalk 才是真正的面向对象语言，但 Smalltalk-80 不支持多继承。

C++从 Simula 继承了类的概念，从 Algol68 继承了运算符重载、引用以及在任何地方声明变量的能力，从 BCPL 获得了//注释，从 Ada 得到了模板、名字空间，从 Ada、Clu 和 ML 取来了异常。

1.1.3　C++中的杰出人物



Dennis M.Ritchie

Dennis M. Ritchie 被称为 C 语言之父，UNIX 之父，生于 1941 年 9 月 9 日，哈佛大学数学博士，现任朗讯科技公司贝尔实验室（原 AT&T 实验室）下属的计算机科学研究中心系统软件研究部的主任一职。他开发了 C 语言，并著有《C 程序设计语言》（The C Programming Language）一书，还和 Ken Thompson 一起开发了 UNIX 操作系统。他因杰出的工作得到了众多计算机组织的公认和表彰，1983 年，获得美国计算机协会颁发的图灵奖（又称计算机界的诺贝尔奖），还获得过 C&C 基金奖、电气和电子工程师协会优秀奖章、美国国家技术奖章等多项大奖。



Bjarne Stroustrup

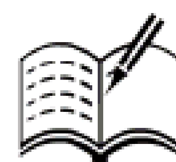
Bjarne Stroustrup 1950 年出生于丹麦，先后毕业于丹麦阿鲁斯大学和英国剑桥大学，是 AT&T 大规模程序设计研究部门负责人，AT&T 贝尔实验室和 ACM 成员。1979 年，Stroustrup 开始开发一种语言，当时称为 C with Class，后来演化为 C++。1998 年，ANSI/ISO C++标准建立，同年，Stroustrup 推出其经典著作《The C++ Programming Language》的第三版。



Scott Meyers

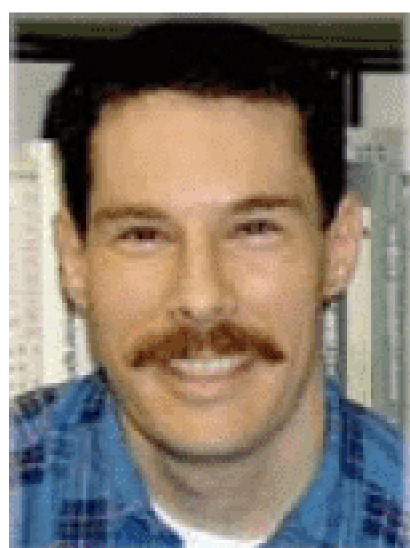
Scott Meyers 是世界顶级的 C++软件开发技术权威人士之一，他拥有 Brown University 的计算机科学博士学位，其著作《Effective C++》和《More Effective C++》很受编程人员的喜爱。Scott Meyers 曾经是 C++Report 的专栏作家，为 C/C++ Users Journal 和 Dr. Dobb's Journal 撰过稿，为全球范围内的客户提供咨询活动。他还是 Advisory Boards for NumeriX LLC 和 InfoCruiser 公司的成员。






Andrei Alexandrescu

Andrei Alexandrescu 被认为是新一代 C++天才的代表人物，2001 年撰写了经典名著《Modern C++Design》，其中对 Template 技术进行了精湛运用，第一次将模板作为参数在模板编程中使用，该书震撼了整个 C++社群，开辟了 C++编程领域的 Modern C++新时代。此外，他还与 Herb Sutter 合著了《C++ Coding Standards》。他在对象拷贝（object copying）、对齐约束（alignment constraint）、多线程编程、异常安全和搜索等领域作出了巨大贡献。


Herb Sutter

Herb Sutter 是 C++ Standard Committee 的主席，作为 ISO/ANSI C++标准委员会的委员，Herb Sutter 是 C++程序设计领域屈指可数的大师之一。他的 Exceptional 系列三本书《Exceptional C++》、《More Exceptional C++》和《Exceptional C++ Style》成为 C++程序员必读书。他是深受程序员喜爱的技术讲师和作家，是 C/C++ Users Journal 的撰稿编辑和专栏作者，曾发表了上百篇软件开发方面的技术文章和论文。他还担任 Microsoft Visual C++架构师，和 Stan Lippman 一道在微软主持 VC 2005（即 C++/CLI）的设计。

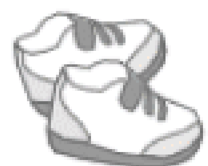

Andrew Koenig

Andrew Koenig 是 AT&T 公司 Shannon 实验室大规模编程研究部门中的成员，同时也是 C++标准委员会的项目编辑，是一位真正的 C++内部权威。Andrew Koenig 的编程经验超过 30 年，其中有 15 年在使用 C++，已经出版了超过 150 篇和 C++有关的论文，并且在世界范围内就这个主题进行过多次演讲，对 C++的最大贡献是带领 Alexander Stepanov 将 STL 引入 C++标准。

1.2　C++语言特点

C++继承了 C 原有的精髓（如高效率、灵活性），增加了对开发大型软件颇为有效的面向对象的机制等，成为一种既可用于表现过程模型，又可用于表现对象模型的优秀的程序设计语言之一。其特点如下：

- ☑ C++与 C 语言相比
  - ➢ 保持与 C 语言的兼容。
  - ➢ 可重用性、可扩充性、可维护性、可靠性都有很大提高。
  - ➢ 支持面向对象的机制。
- ☑ C++与其他面向对象语言相比
  - ➢ 可读性更好，可直接在程序中映射问题空间的结构。



- 代码质量高，比其他面向对象语言执行效率高得多。

☑ C++与 C 不同之处

- C 源程序文件扩展名为.c，而 C++为.cpp。
- 在 Windows 下，当给定扩展名为.c 时，启动 C 的编译器；而当给定扩展名为.cpp 时则启动 C++的编译器。
- 在 Linux 下，后缀名字只是给人看的，编译时需要制定编译器 g++ main.cpp。



Note

1.3 Visual Studio 2010 集成编译环境

使用 C++的开发环境有很多种，如常见的 Visual C++ 6.0 等。Visual Studio 2010 是微软继 Visual C++ 6.0 之后新设计的集成开发环境，它更加支持 C++标准规范，对新标准——C++0x 提供全面的支持。下面将介绍它的使用方法。

1.3.1 安装 Visual Studio 2010

在安装 Visual Studio 2010 之前，首先要了解其安装必备条件，检查计算机的软硬件配置是否满足安装 Visual Studio 2010 开发环境的要求，具体要求如表 1.1 所示。

表 1.1 安装 Visual Studio 2010 所需的必备条件

软 硬 件	描 述
处理器	1.6GHz 处理器，建议使用 2.0GHz 双核处理器
RAM	1GB，建议使用 2GB 内存
可用硬盘空间	系统驱动器上需要 5.4GB 的可用空间，安装驱动器上需要 2GB 的可用空间
CD-ROM 驱动器或 DVD-ROM	必须使用
显示器	建议使用 1024×768，增强色 16 位
鼠标	微软鼠标或兼容的指针设备
操作系统及所需补丁	Windows XP（SP3）、Windows Server 2003（SP2）、Windows Vista、Windows 7



注意：

Windows XP Home 不支持本地 Web 应用程序开发，只有 Windows 专业版和服务器版才支持本地 Web 应用程序开发。

下面将详细介绍如何安装 Visual Studio 2010，使读者掌握每一步的安装过程，阅读本节之后，读者完全可以自行安装 Visual Studio 2010。安装 Visual Studio 2010 的步骤如下：

（1）将 Visual Studio 2010 安装盘放到光驱中，光盘自动运行后会进入安装程序界面，如果光盘不能自动运行，可以双击 setup.exe 可执行文件，应用程序会自动跳转到如图 1.1 所示的“Microsoft Visual Studio 2010 安装程序”界面。该界面上有两个安装选项，分别为安装 Microsoft





Note

Visual Studio 2010 和检查 Service Release，一般情况下需安装第一项。

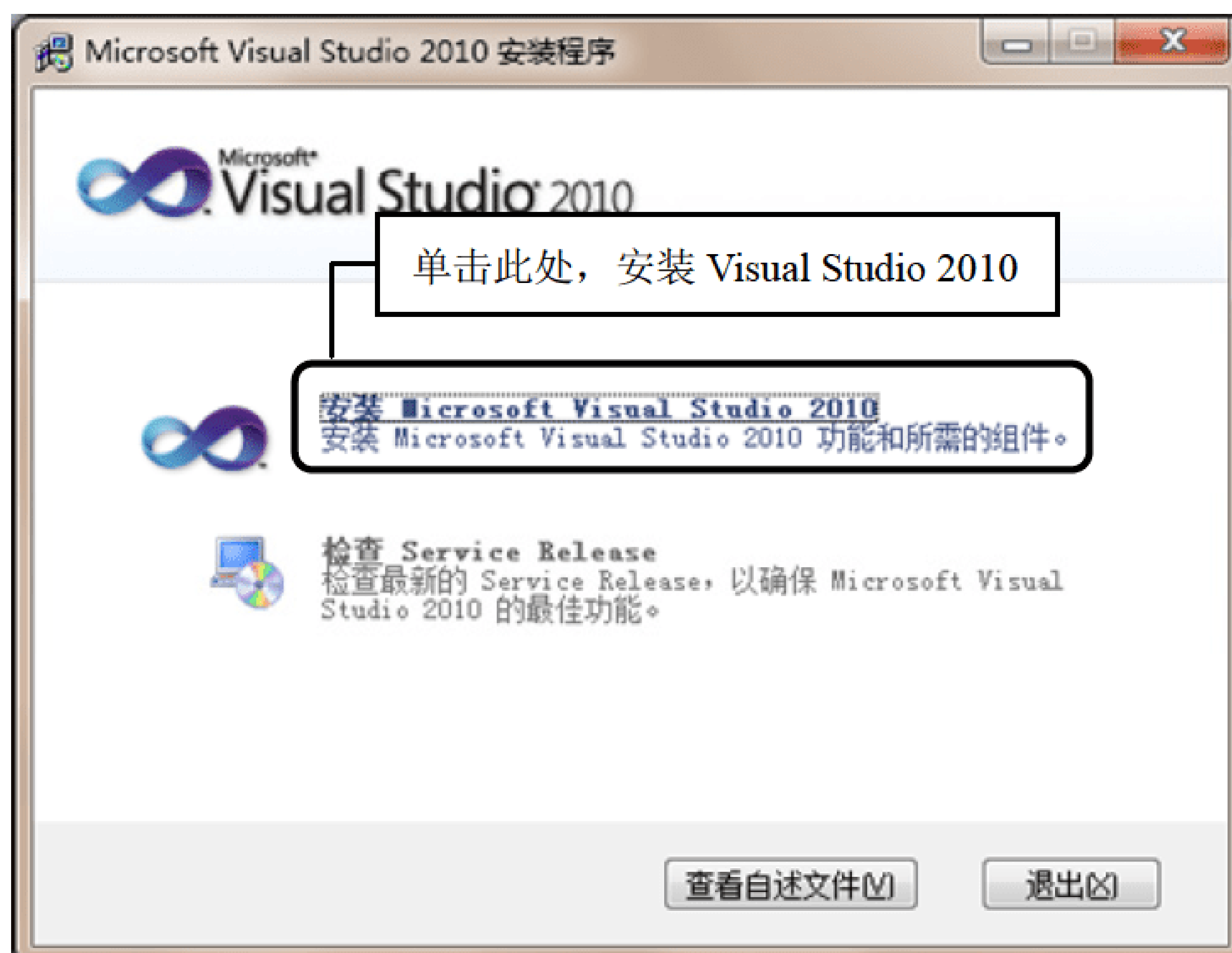


图 1.1　Visual Studio 2010 安装界面

（2）单击第一个安装选项“安装 Microsoft Visual Studio 2010”，弹出如图 1.2 所示的 Visual Studio 2010 安装向导界面。

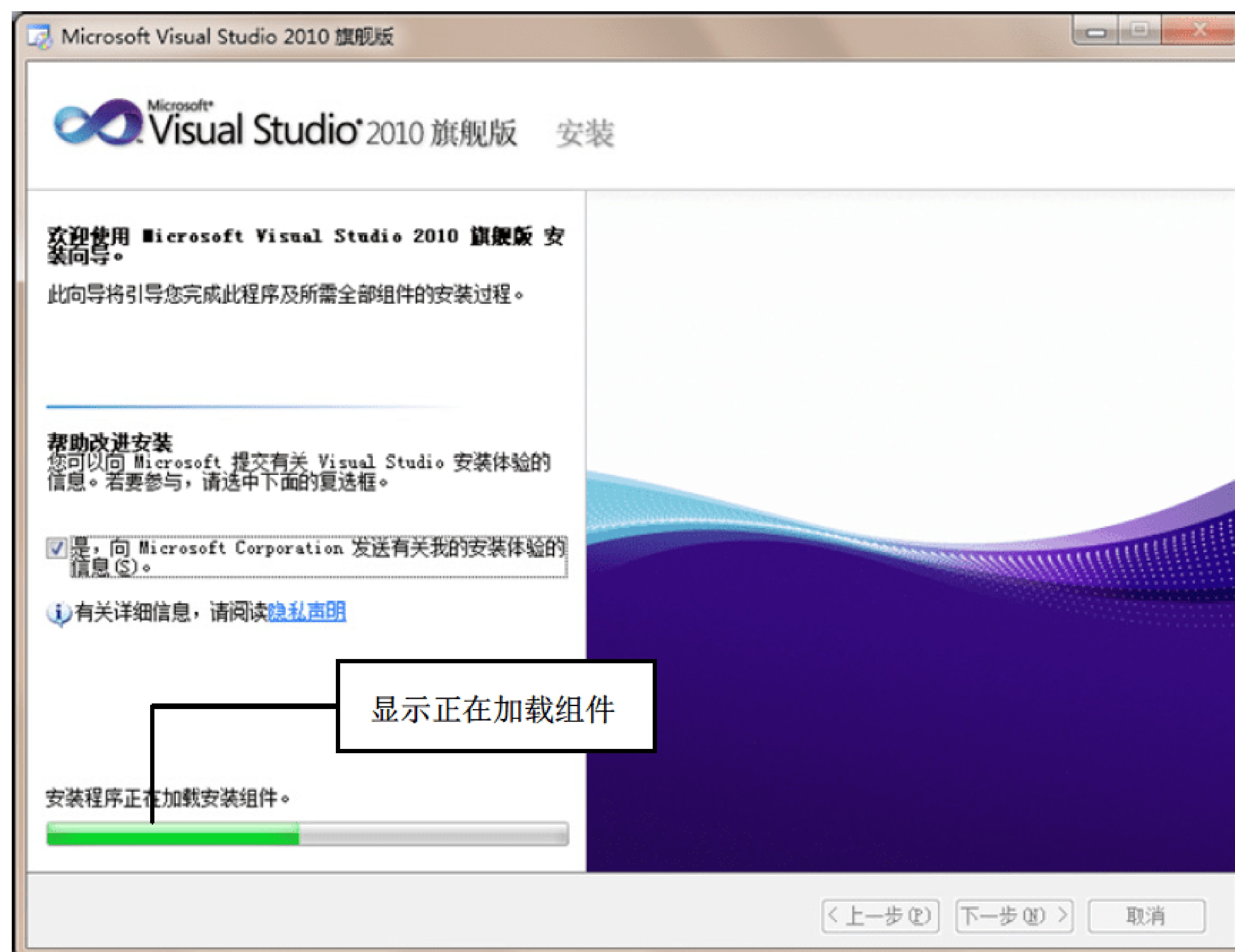


图 1.2　Visual Studio 2010 安装向导





（3）单击“下一步”按钮，弹出如图 1.3 所示的“Microsoft Visual Studio 2010 安装程序-起始页”界面，该界面左侧显示的是关于 Visual Studio 2010 安装程序所需的组件信息，右边显示用户许可协议。

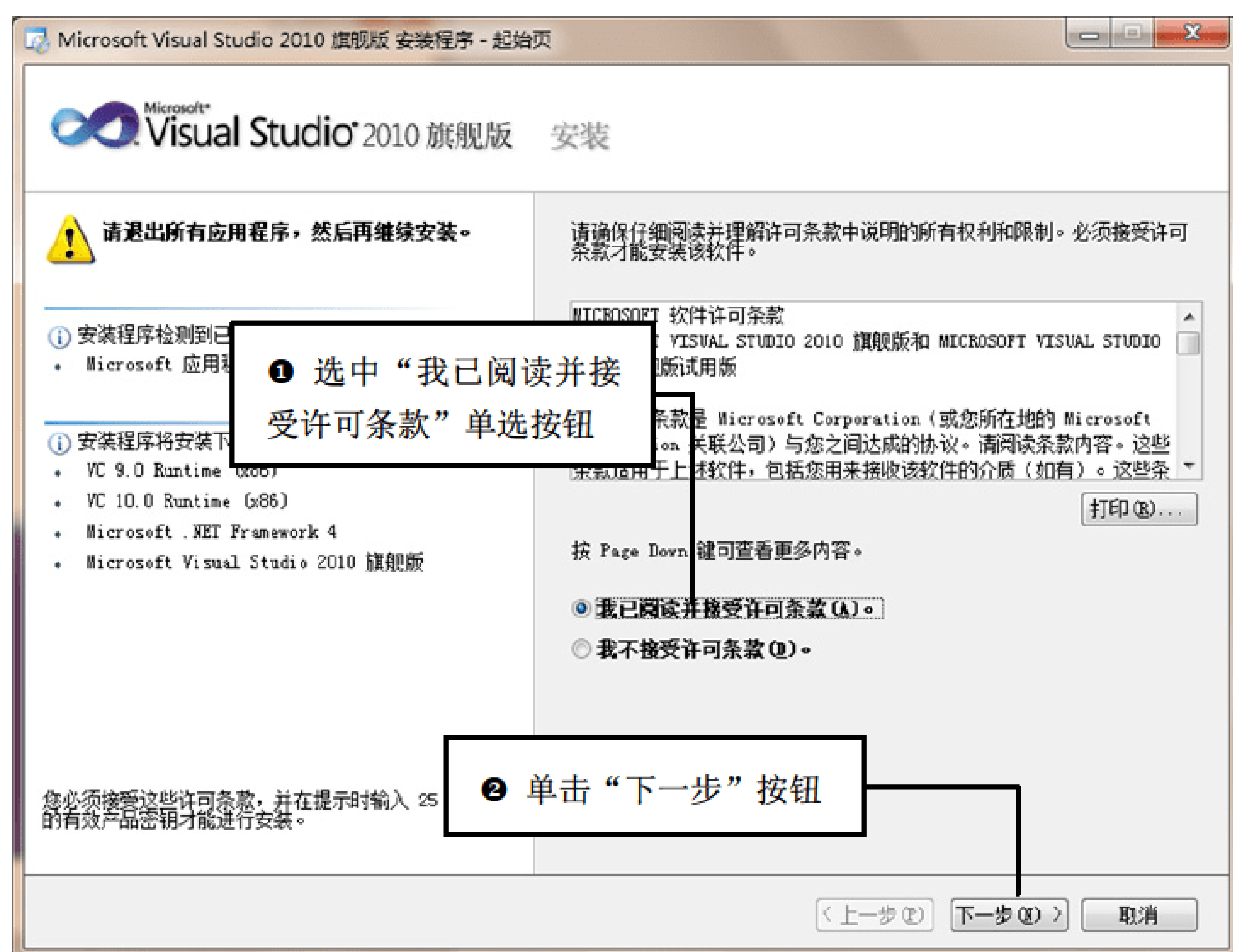


图 1.3　Visual Studio 2010 安装程序-起始页

（4）选中“我已阅读并接受许可条款”单选按钮，单击“下一步”按钮，弹出如图 1.4 所示的“Microsoft Visual Studio 2010 安装程序-选项页”界面，用户可以选择要安装的功能和产品安装路径，一般使用默认设置即可，产品默认路径为“C:\Program Files\Microsoft Visual Studio 10.0\”。在本程序中的安装路径为“D:\Program Files\Microsoft Visual Studio 10.0\”。

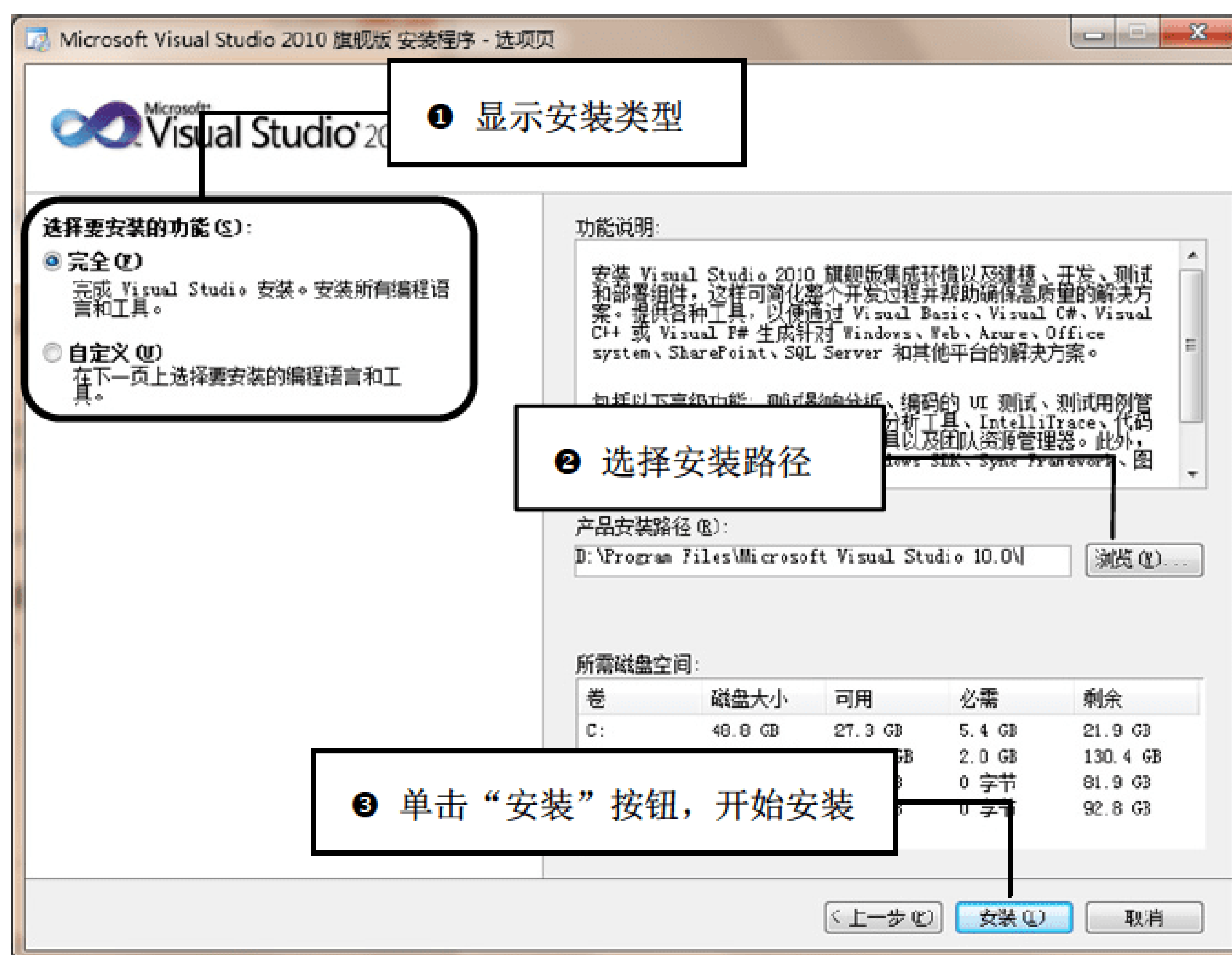


图 1.4　Visual Studio 2010 安装程序-选项页

在选择安装选项页中，用户可以选择“完全”和“自定义”两种方式。如果选择“完全”，程序会安装系统的所有功能，如图 1.5 所示。如果选择“自定义”，用户可以选择希望安装的项目，增加了安装程序的灵活性，如图 1.6 所示。

❶ 选择“完全”安装方式

❷ 选择安装路径

❸ 单击“安装”按钮，开始安装

图 1.5　选择“完全”安装方式

❶ 选择“自定义”安装方式

❷ 选择安装路径

❸ 单击“安装”按钮，开始安装

图 1.6　选择“自定义”安装方式

（5）在图 1.5 中，选择好产品安装路径后单击“安装”按钮，进入产品的安装界面。在图 1.6 中，选择好产品安装路径后单击“下一步”按钮，进入选择要安装的功能界面，如图 1.7 所示。

（6）选择好产品安装路径之后，单击“安装”按钮，进入如图 1.8 示的“Visual Studio 2010 安装程序-安装页”界面，显示正在安装组件。



Note

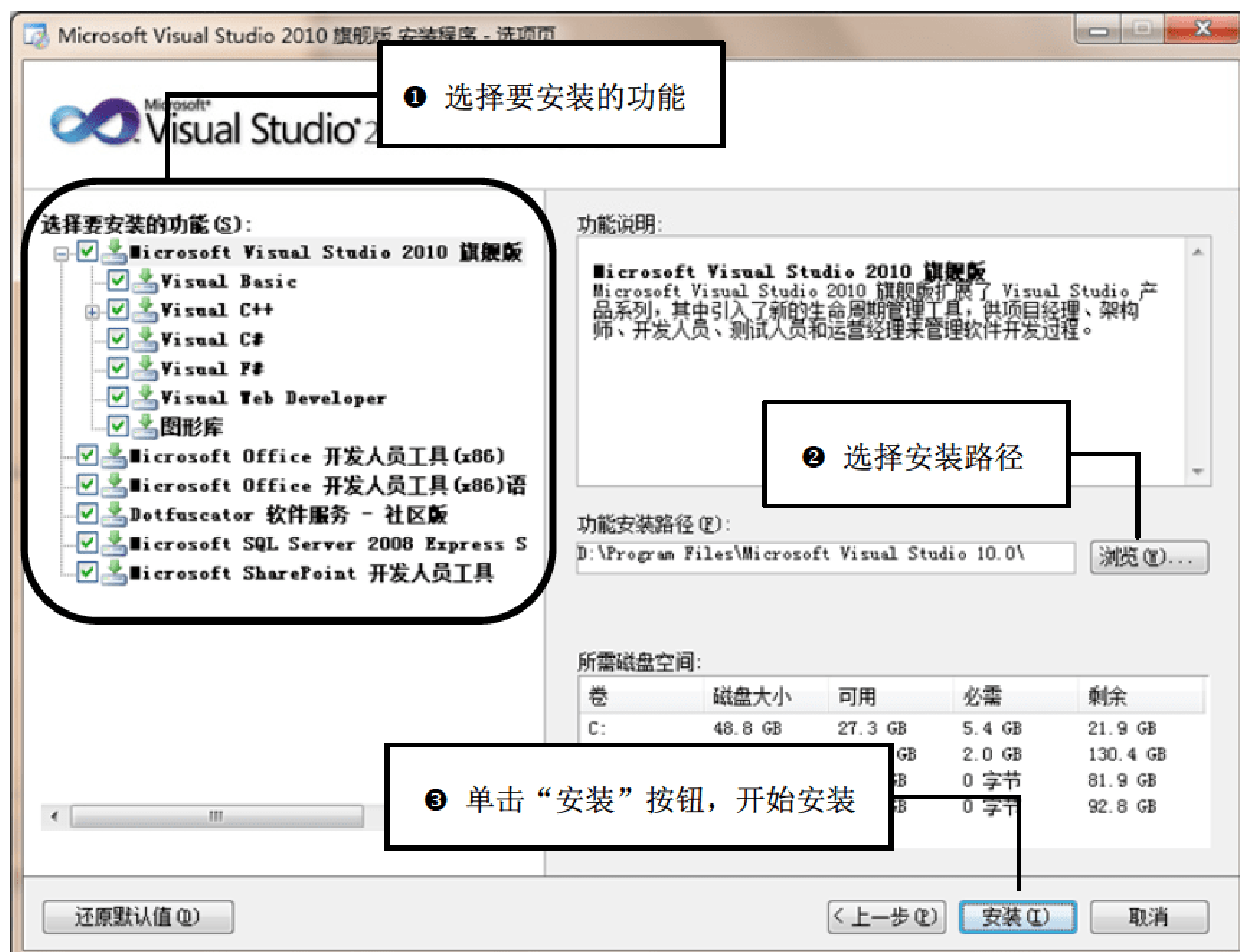


图 1.7　选择安装的功能



图 1.8　Visual Studio 2010 安装程序-安装页

（7）安装完毕后，单击“下一步”按钮，弹出如图 1.9 示的“Visual Studio 2010 安装程序-完成页”界面，单击“完成”按钮，至此，Visual Studio 2010 程序开发环境安装完成。



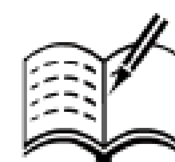




图 1.9　Visual Studio 2010 安装程序-完成页

1.3.2　卸载 Visual Studio 2010

如果想卸载 Visual Studio 2010，可以按以下步骤进行：

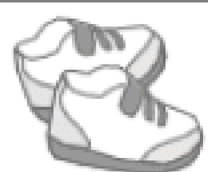
（1）在 Windows 7 操作系统中，打开“控制面板”/“程序”/“程序和功能”，在打开的窗口中选中“Microsoft Visual Studio 旗舰版-简体中文”，如图 1.10 所示。



图 1.10　添加或删除程序

（2）单击“卸载/更改”按钮，进入 Microsoft Visual Studio 2010 安装程序维护模式，如图 1.11 所示。

（3）单击“下一步”按钮，进入 Microsoft Visual Studio 2010 安装程序-维护页，如图 1.12 所示，单击“卸载”按钮进行卸载。





Note



图 1.11　Microsoft Visual Studio 2010 安装程序维护模式

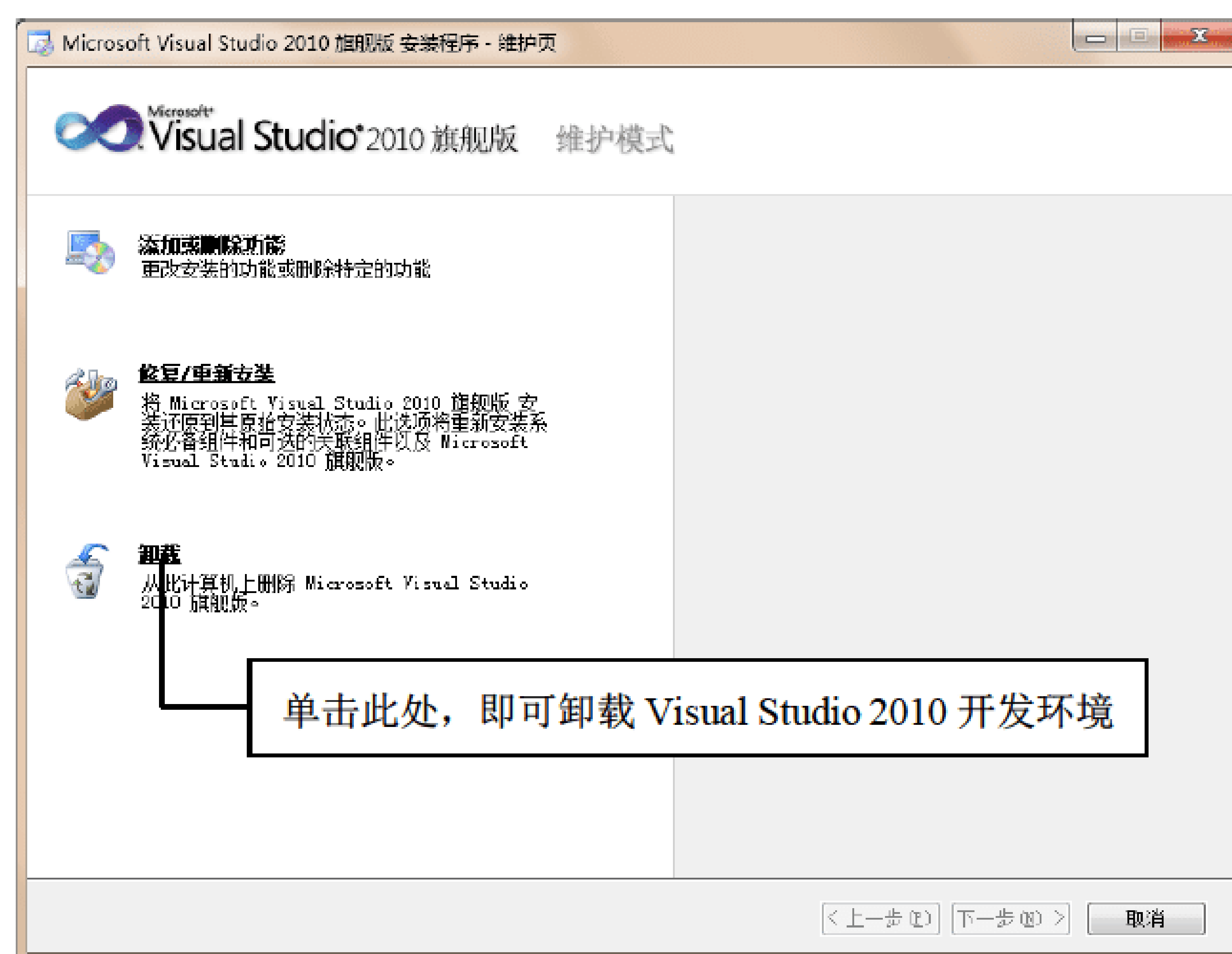


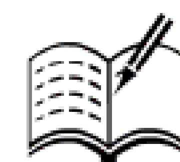
图 1.12　Microsoft Visual Studio 2010 安装程序-维护页

1.3.3　使用 Visual Studio 2010 创建一个 C++控制台程序

创建项目的过程非常简单，首先启动 Visual Studio 2010 开发环境，选择“开始”/“程序”/Microsoft Visual Studio 2010/Microsoft Visual Studio 2010 命令，即可进入 Visual Studio 2010 开发环境，其右侧会列出已安装的产品，如图 1.13 所示。

Microsoft Visual Studio 2010 的起始页界面如图 1.14 所示。





Note



图 1.13　进入 Visual Studio 2010 开发环境



图 1.14　Visual Studio 2010 起始页

启动 Visual Studio 2010 开发环境之后，可以通过两种方法创建项目：一是选择“文件”/“新建”/“项目”命令，如图 1.15 所示；二是通过在起始页中选择“新建项目”选项，如图 1.16 所示。

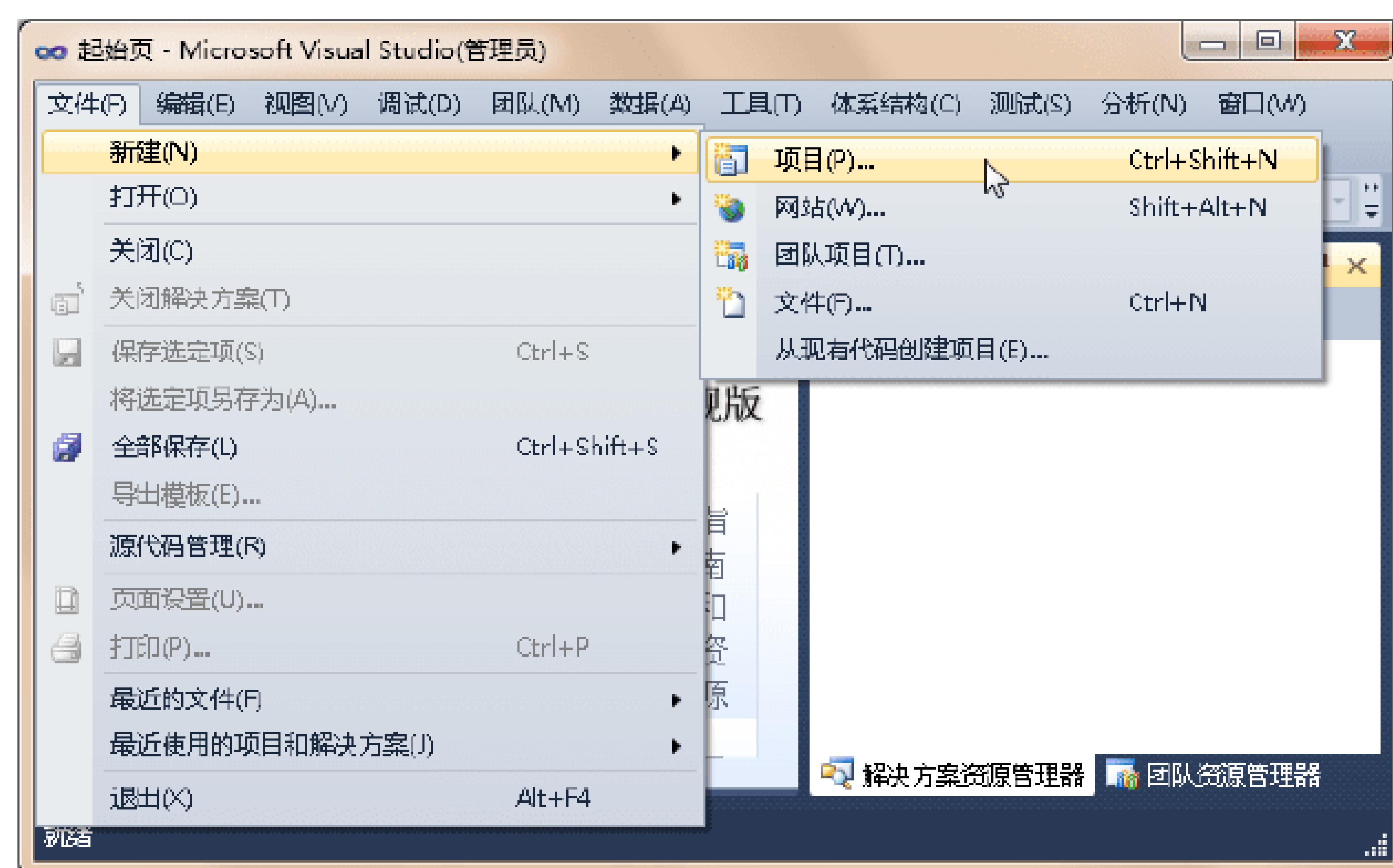
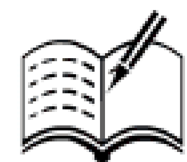


图 1.15　选择命令创建项目



Note

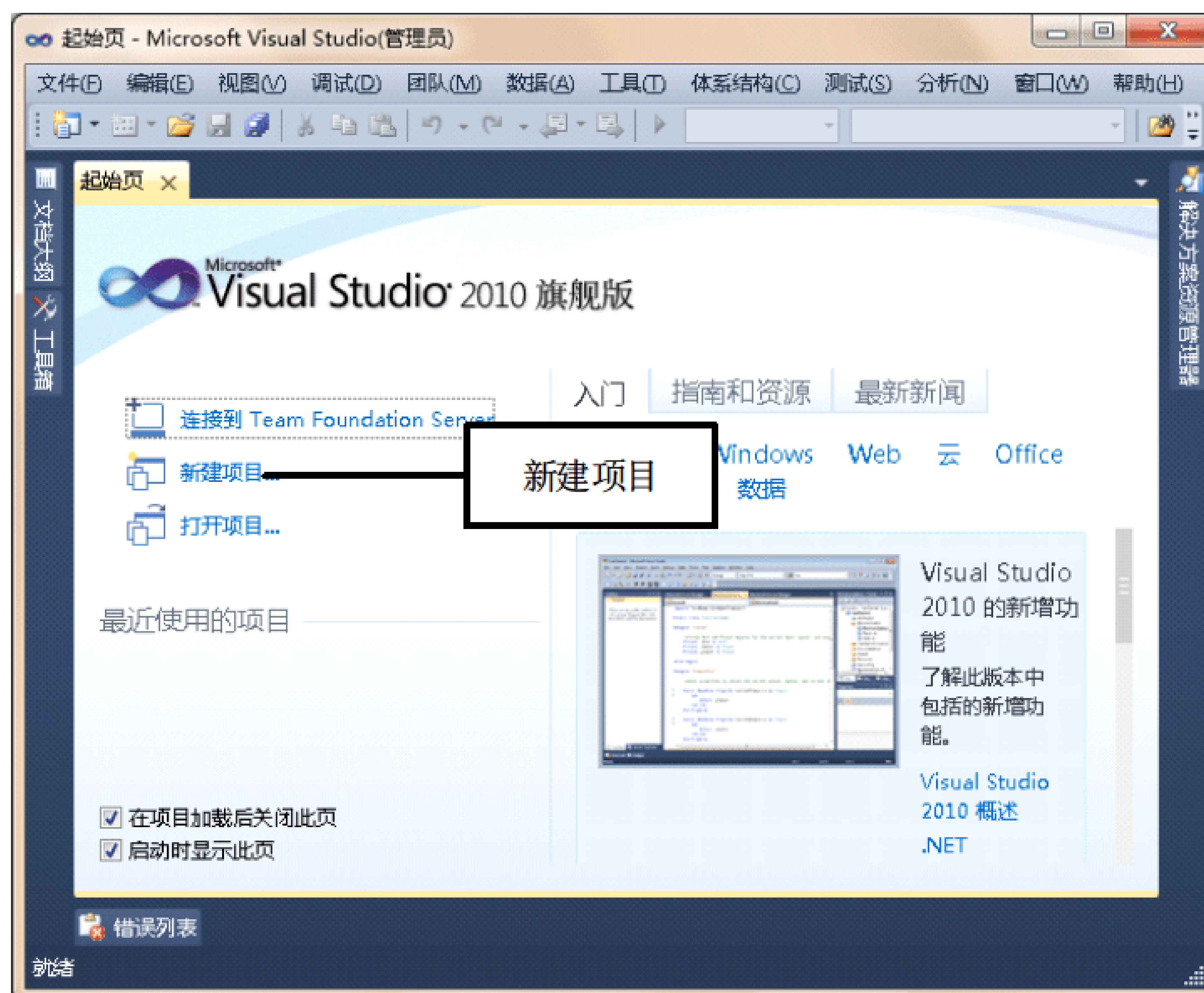


图 1.16　在起始页创建项目

选择其中一种方法创建项目，将弹出如图 1.17 所示的“新建项目”对话框。

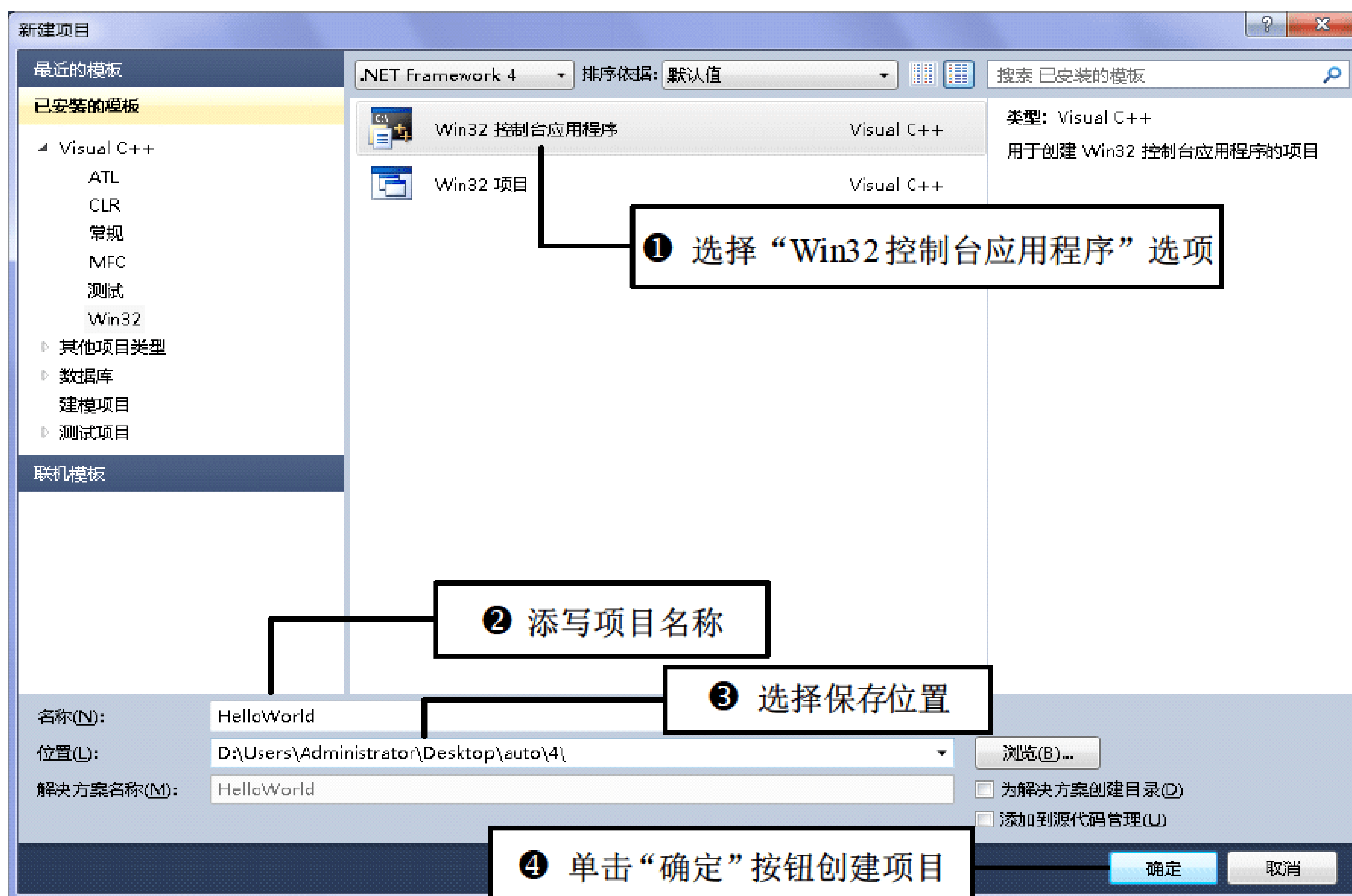


图 1.17　“新建项目”对话框

在图 1.17 中选择 Win32，再选择 Win32 控制台程序后，用户可对所要创建的项目进行命名、选择保存的位置、是否创建解决方案目录的设定，在命名时可以使用用户自定义的名称，也可使用默认名，用户可以单击“浏览”按钮设置项目保存的位置。需要注意的是，解决方案名称与项目名称一定要统一，然后单击“确定”按钮。





在应用程序向导中单击“完成”按钮，接受当前设置完成创建，如图 1.18 所示。详细设置将在后面章节讲解，这里选择默认设置，进入项目。

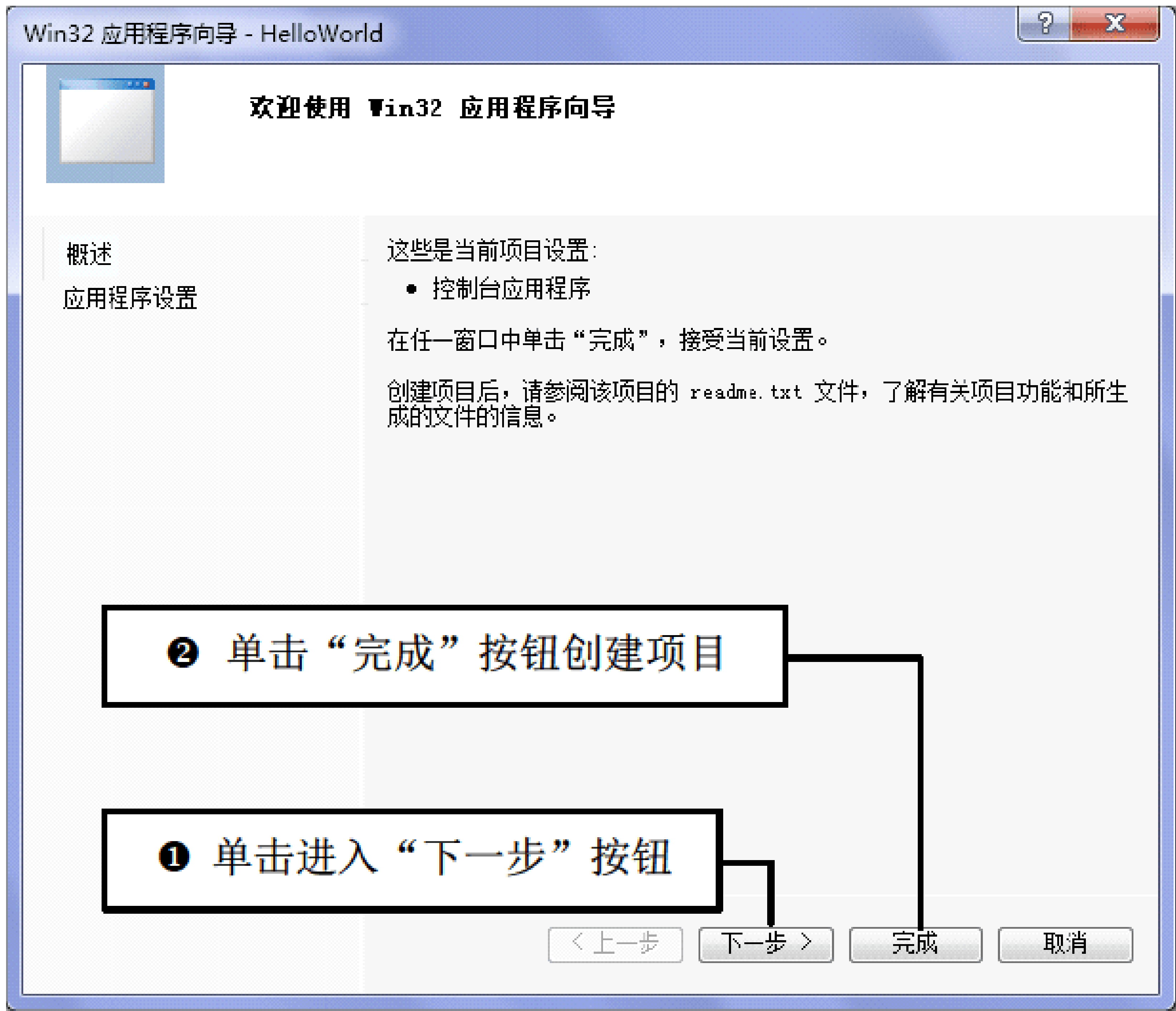


图 1.18　应用程序向导

1.3.4　编写第一个 C++程序“Hello World!”

下面就来写一个简单的小程序。

【例 1.1】　HelloWorld 程序演示。

实例位置：光盘\MR\Instance\01\1.1

编写一个输出“HelloWorld!”程序。

```
#include "stdafx.h"
int main(int argc,_TCHAR* argv[])                    //主函数
{
      printf("HelloWorld!");                        //一个完整的语句需要后面加分号
      return 0;
}
```

按 F7 键编译程序，如图 1.19 所示。

直接运行这个程序（按 Ctrl+F5 快捷键），如图 1.20 所示。

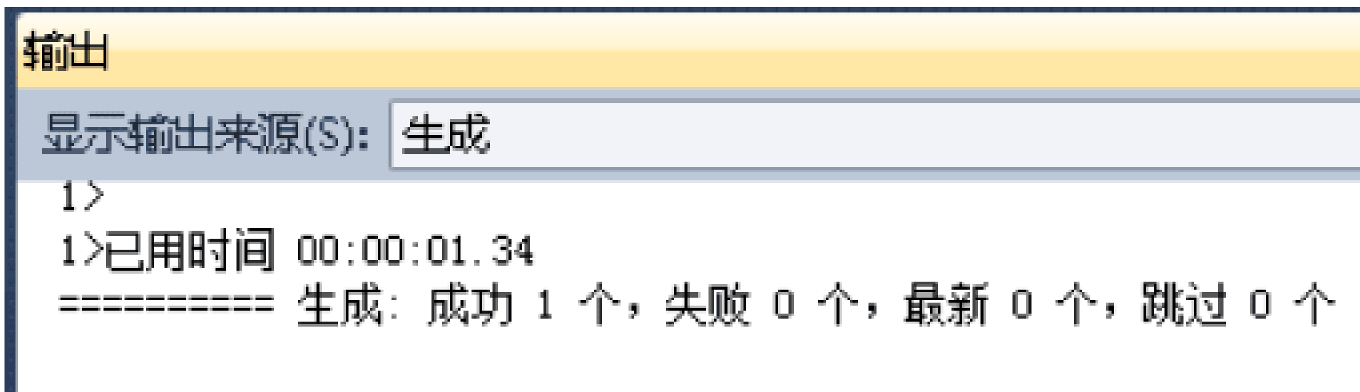


图 1.19　编译 HelloWorld 程序

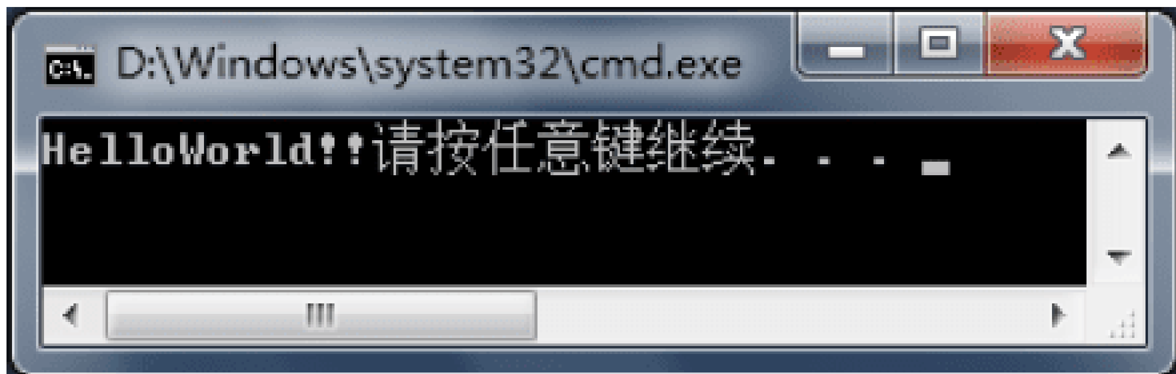
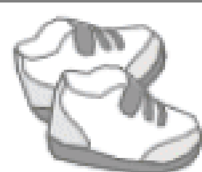
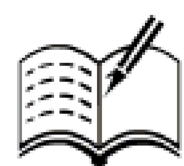


图 1.20　HelloWorld 程序



1.4 本书代码使用指南



Note

本书在各个章节的实例部分都包含了代码，若想运行它们有两种办法：一种是根据路径提供打开本书所提供的项目（.sln 文件），如图 1.21 所示。另一种是在电脑中通过本书提供的路径查找到项目文件夹，将相应的源文件和头文件拖曳复制到项目资源管理器中。

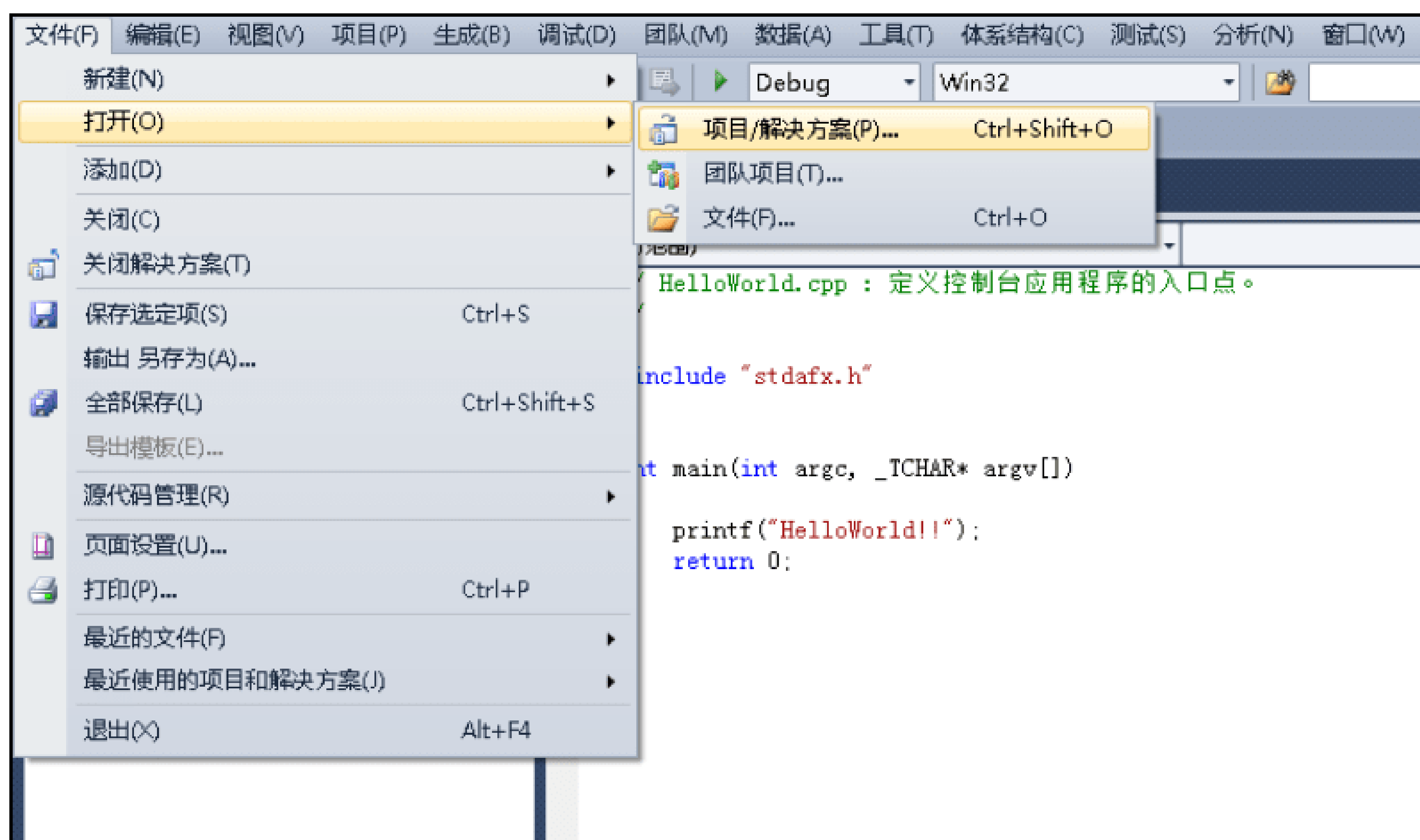


图 1.21 打开已有的工程

1.5 本 章 小 结

本章简单介绍了 C++语言的发展历程、主要优势和集成环境 Visual Studio 2010 的使用方法。

第 2 章

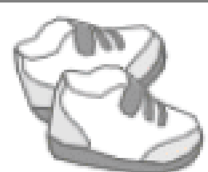
认识 C++程序

（视频讲解：54 分钟）

在开始系统地学习编程之前，可以先通过一个 C++程序领略一下 C++的神奇之处。函数是完成一定功能的执行代码段，是 C++中的重要概念，也是程序设计的重要手段。

本章能够完成的主要范例（已掌握的在方框中打勾）

- ☐ 掌握 C++第一个编程例子
- ☐ C++的基本组成
- ☐ 初步了解函数
- ☐ 了解 C++的编程规范



2.1　我的第一个 C++程序



Note

2.1.1　创建第一个 C++程序

用编程语言编写程序的完整流程应该分为以下 7 个步骤：

（1）定义一个程序目标。

（2）设计程序。

（3）编写代码。

（4）编译。

（5）运行程序。

（6）测试和调试。

（7）程序维护。

下面就来写一个简单的小程序。

如图 2.1 所示，项目中左边的解决方案管理器中显示了本程序所有包含和依赖的文件。

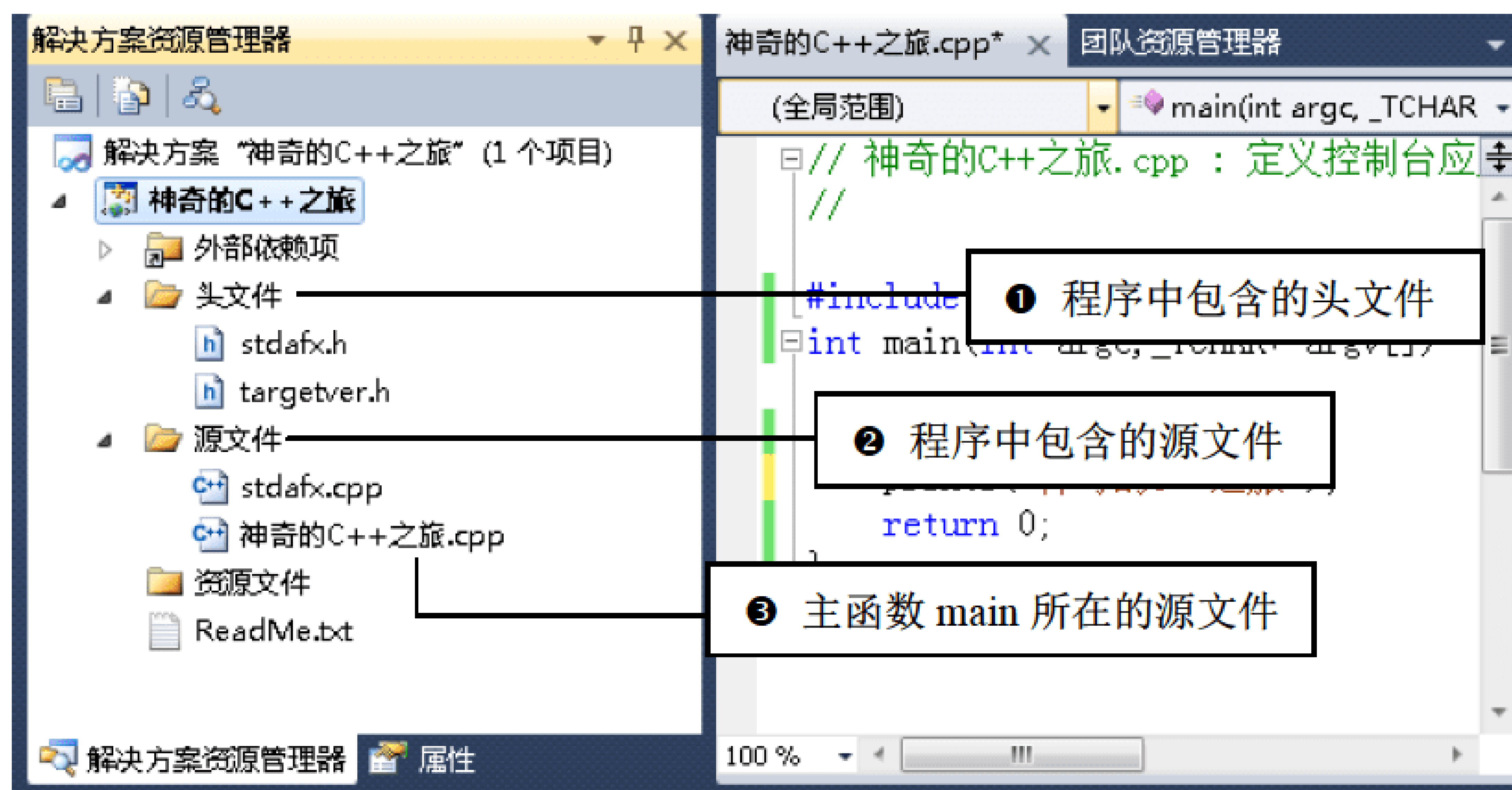


图 2.1　解决方案资源管理器

头文件存储着函数、变量的声明，与之相对应的源文件提供了这些函数。本项目的入口在“神奇的 C++之旅. cpp”这个源文件中，因为它包含着程序的入口主函数 main。

C++程序的入口是 main 函数，控制台应用程序也可以用_tmain 来作为入口。为保持一致，今后将把_tmain 函数改为 main。但无论哪个，编译器都会找到它们作为入口使用。在这个源文件中包含着一个头文件 stdafx.h，它由编译器生成，其中包含了项目中常用的头文件。return 语句表示函数结束，返回相应的值。在主函数中执行 return 后，程序结束。双斜杠后边的绿色文字叫做注释，对程序只起到解释和说明。程序运行时，注释不会被当作代码来编译，如图 2.2 所示。





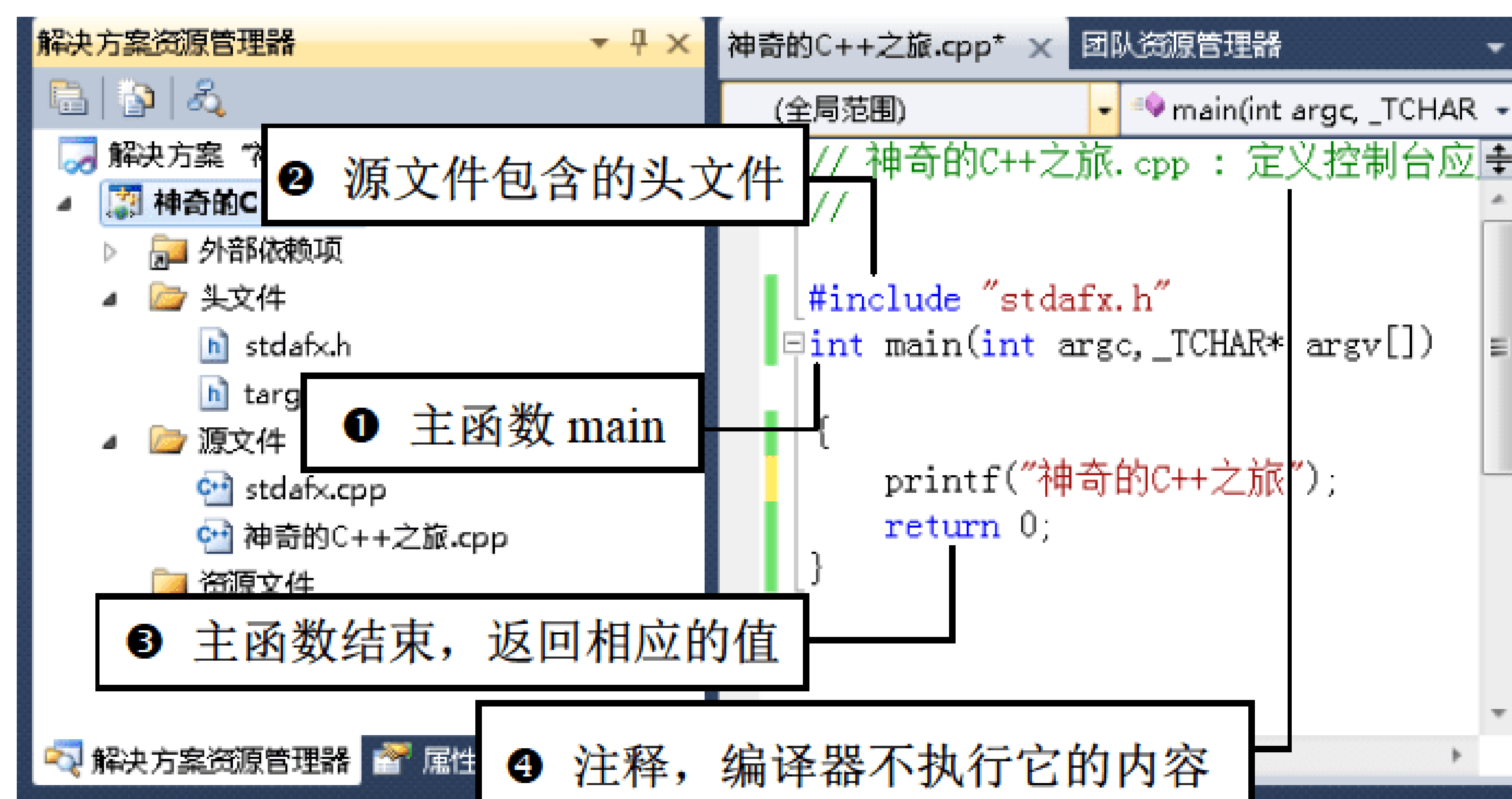


图 2.2　神奇的 C++之旅.cpp 源文件

【例 2.1】　“神奇的 C++之旅”程序演示。

实例位置：光盘\MR\Instance\02\2.1

下面编写一个“神奇的 C++之旅”程序，如图 2.3 所示。

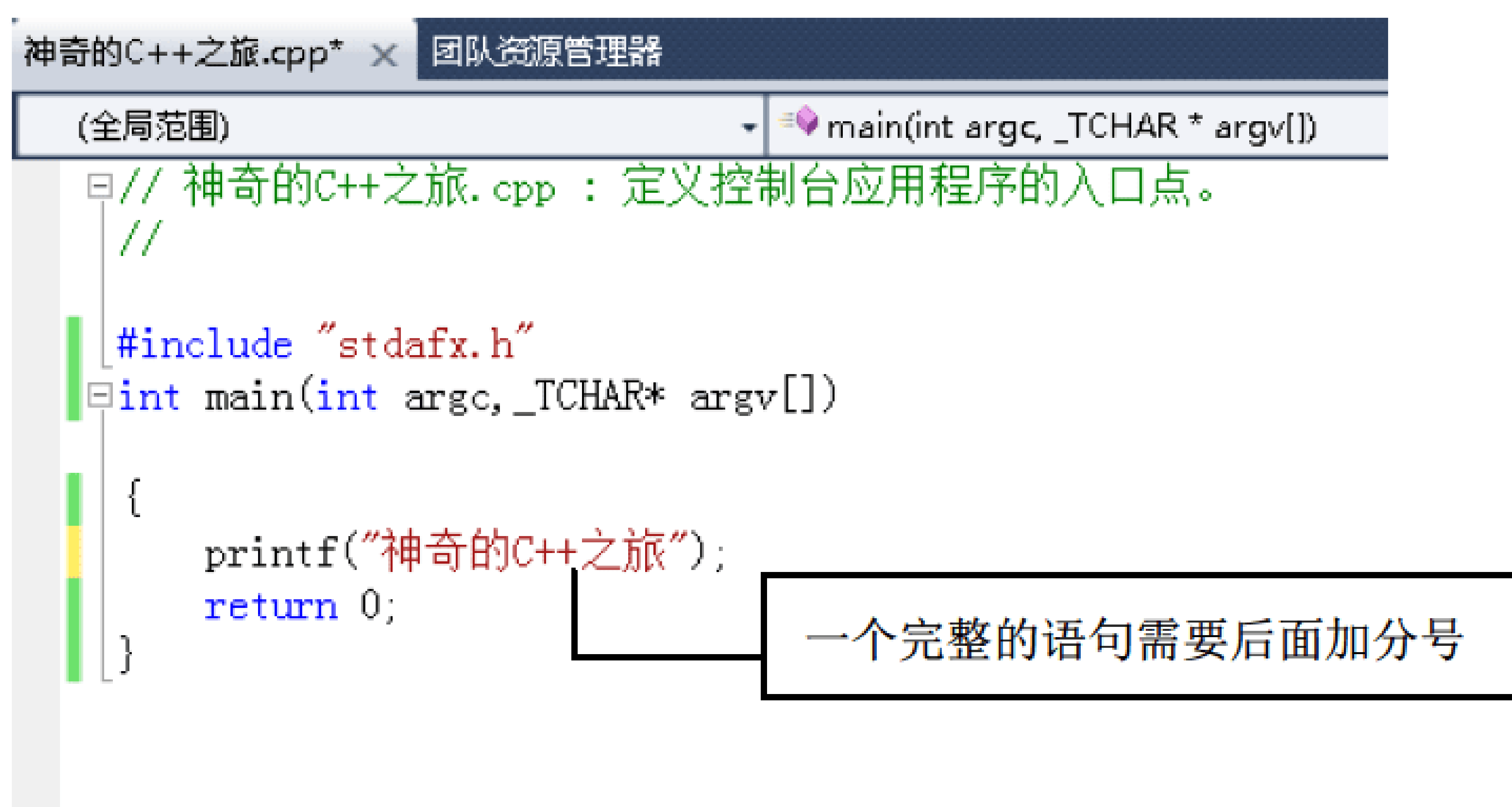


图 2.3　“神奇的 C++之旅”程序

按 F7 键编译程序，如图 2.4 所示。

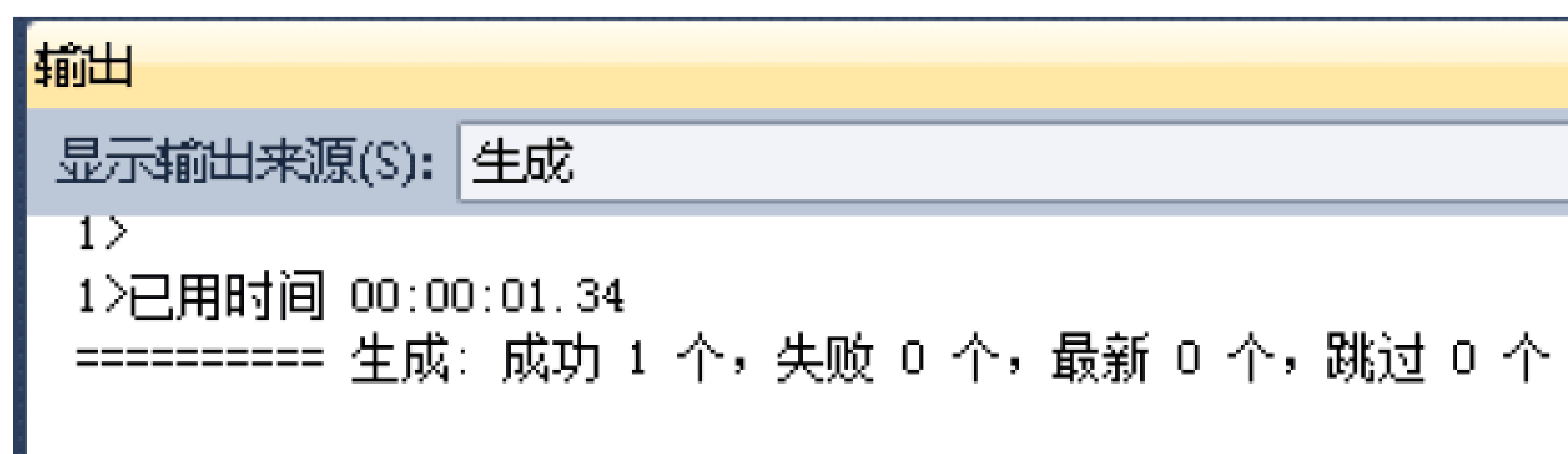


图 2.4　编译神奇的 C++之旅程序

“调试”菜单中有“启动调试”和“开始执行”两个命令，如图 2.5 所示。调试运行时会查找程序中的错误，并在设置的断点处进行停留。开始执行则不会进行调试，直接运行程序，当程序遇到编译错误时，执行失败。现在，按 Ctrl+F5 快捷键直接运行这个程序，如图 2.6 所示。



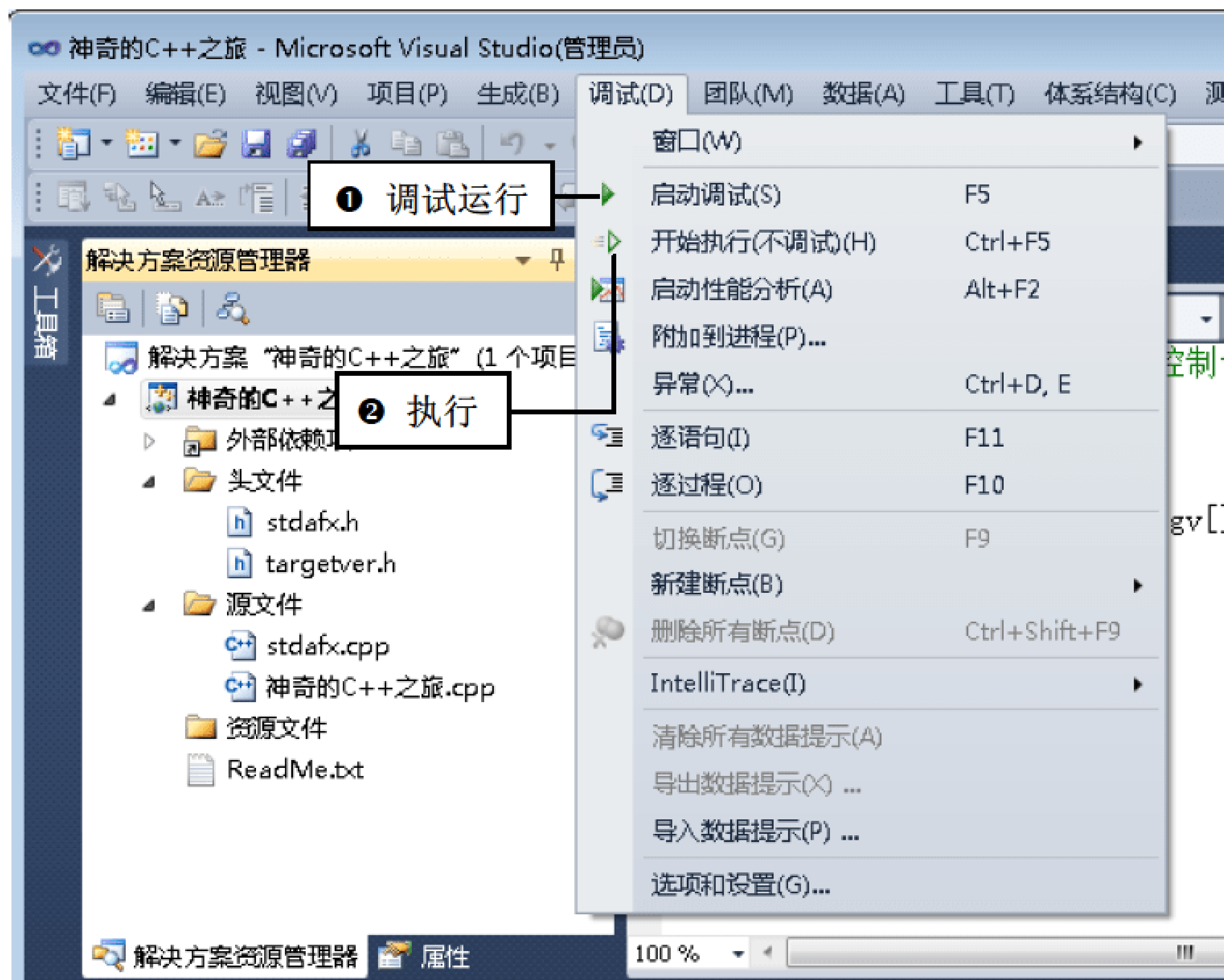


图 2.5　运行程序

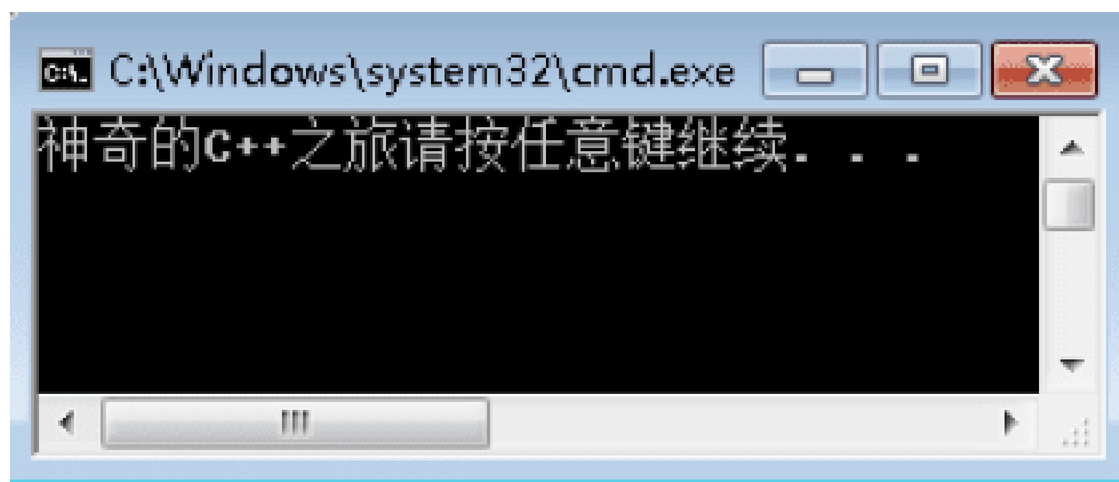


图 2.6　神奇的 C++之旅程序



注意：

如果运行的输出结果与用户期望的不一样，那么就需要对该源程序进行功能测试，以找出逻辑上的错误。

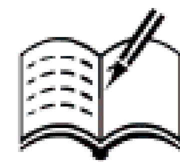
前边的“神奇的 C++之旅”是通过输出语句 printf 输出的，后边的“请按任意键继续”的提示语是控制台应用程序添加上去的。printf 是标准输出语句，双引号内的字符原样输出，更改双引号里的内容就可以输出不同的语句。这里如果有格式控制（如%d），后面需要有相对应的输出项，后面章节将会详细介绍。

2.1.2　C++程序的基本组成

一般来说，一个标准的 C++程序通常由预处理命令、函数、语句、变量、输入/输出及注释等几部分组成，下面分别进行介绍。



- ☑ 预处理命令：在 C++程序中，预处理命令以“#”开头。C++提供了 3 种预处理命令，分别为宏定义命令、文件包含命令及条件编译命令。
- ☑ 函数：一个 C++程序通常由若干个函数组成，这些函数可以是 C++系统提供的库函数，也可以是用户根据需要编写的自定义函数。在这些函数中，必须有且仅有一个主函数 main，不论主函数位于什么位置，该程序都是从主函数开始执行的。
- ☑ 语句：是组成对象的基本单元，它包含顺序语句、选择语句、循环语句等。所有的语句以分号结束，最简单的语句是空语句，它仅包括一个分号。
- ☑ 变量：在 C++程序中，需要将数据存放于内存单元中，而变量就是用来存储和访问内存单元中数据的标识符。变量有整型、字符型、浮点型等基本数据类型。
- ☑ 输入/输出语句：在 C++程序中，经常要使用到输入和输出语句，用于接收用户的输入及返回程序运行结果。
- ☑ 注释：可以帮助读者阅读源程序，但并不参与程序的运行。



2.2　C++的常用概念

2.2.1　预处理命令

C++的程序中带“#”号的语句被称为宏定义或预编译指令，C++中的语句、宏定义和预编译指令会在后面章节中讲到。#include 在代码中是包含和应用的意思，#include"stdefx.h"就是说明代码要引用 stdefx 文件内容，编译器在编译程序时会将 stdefx 中的内容在#include"stdefx.h"展开。

2.2.2　注释

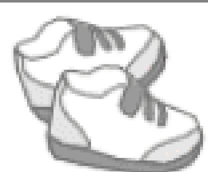
代码注释用来禁止语句的执行，编译器不会对注释的语句进行编译。C++中有两种注释方法，其中“//”是单行注释，单行注释只能注释符号“//”后面的内容，到本行代码结束的位置结束；“/**/”是多行注释，多行注释的使用方法是符号“/*”放在将要注释代码的前面，符号“*/”放在将要注释代码的末尾，符号“/*”和“*/”中间的内容就会被注释。另外，多行注释中不允许嵌套多行注释，例如“/*/**/*/”，最后出现的“*/”符号将会无效。

（1）单行注释

```
//要注释的内容
```

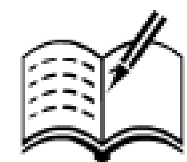
（2）多行注释

```
/*注释内容的开始
…
注释内容的结束*/
```



在第一个 C++程序中加入注释，代码如下：

```
#include "stdafx.h"                              //头文件引用
int main(int argc,_TCHAR* argv[])                //主函数
{
    printf("神奇的 C++之旅");                    //执行输出
    return 0;                                    //返回值
}
```



Note

开发人员可以在代码中加入注释，用来说明代码的用意，可以方便日后自己或者别人查看。

2.2.3 main 函数

单词 main 代表主函数的意思，main 函数是程序执行的入口，程序从 main 函数的第一条指令开始执行，直到 main 函数结束，整个程序也将执行结束。注意函数的格式单词 main 后面有个小括号“()”，小括号内是放参数的地方。函数相关的内容将在后面章节中讲到。

2.2.4 函数体

大括号“{}”中的内容是需要执行的内容，称为函数体，函数体是按照代码的先后顺序执行的，写在前面的代码先执行，写在后面的代码后执行。代码“printf("神奇的 C++之旅");”表示通过输出流输出语句“神奇的 C++之旅”，语句“神奇的 C++之旅”两边的双引号代表此语句是字符串常量，printf 表示输出。

2.2.5 函数返回值

void 表示函数的返回值，函数的返回值是用来判断函数执行情况以及返回函数执行结果的。void 代表不返回任何数据，如果要返回还需要使用 return 语句。

2.3 初步了解函数

2.3.1 一个简单的函数

函数又叫方法，即实现某项功能或服务的代码块。例如，要实现输出一行文字的功能，编写输出函数如下：





Note

```
void show()
{
    printf("神奇的 C++之旅");                    //执行输出
}
```

void 表示该函数仅有执行功能，没有返回值；如果是 int，则表示该函数返回一个整数，这个返回的整数将为其他函数所使用；show 为该函数的名字，后面的小括号列出该函数需要的参数，假如没有列出函数，则表示该函数不需要参数。

函数主体从左大括号“{”开始，到右大括号“}”结束，中间是该函数的功能，如输出一行字符“神奇的 C++之旅”。

上面的程序段即为我们定义了一个非常简单的函数，是个以 show 命名的函数，其作用是输出“神奇的 C++之旅”。

当需要调用该函数时就可以写：

```
show();
```

这样，程序就会跳转到该函数的定义部分去执行，当函数执行完毕后，再跳回到原始位置继续往下执行，这就好像是我们正在看书，突然听见有人敲门，这时就需要去开门，开完门之后回来继续看书。

【例 2.2】 main 函数。

实例位置：光盘\MR\Instance\02\2.2

```
#include "stdafx.h"
void show()                                      //定义 show 函数
{
    printf("神奇的 C++之旅");                    //输出语句
}
int main(int argc,_TCHAR* argv[])                //主函数
{
    printf("main 函数开始");                      //输出 main 函数开始
    show();                                       //执行 show 函数
    printf("main 函数结束");                      //输出 main 函数结束
    return 0;                                     //返回值
}
```

运行程序，显示效果如图 2.7 所示。

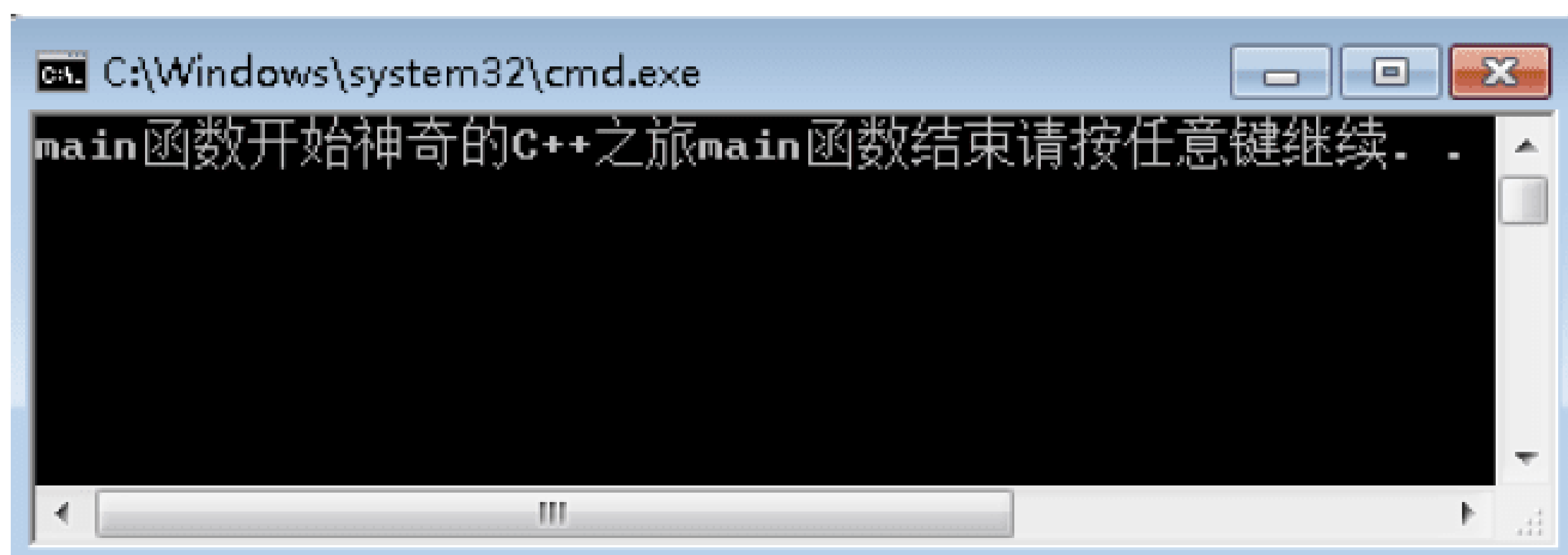


图 2.7　执行结果





注意：

一般情况下，普通函数需要被激活或者调用才能起作用，而 main 函数可由操作系统直接调用。



Note

2.3.2　函数的传参

在 2.3.1 节中，为了便于读者了解函数，使用了一个不带参数的函数：

```
void show()                                   //定义 show 函数
{
    printf("神奇的 C++之旅");                  //输出语句
}
```

由于在 show 函数内仅仅是输出一行字符，并未涉及任何运算，因此小括号中的参数为空，假如有两个变量 a 和 b，它们的值分别为 3 和 4，如果要在 show 函数中将两个变量相加，那么就需要为 show 函数提供两个参数，如下：

```
int show(int a,int b)
{
    return a+b;
}
```

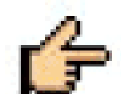
这两个参数是形式上的参数，即它表示 show 函数可以接收两个整数，并用两个形式上的名字来“称呼”这两个整数（形式上的名字不是实际的变量名），它们只用来代表两个整数的操作，一个是 a，另一个是 b，然后在函数体中将 a 和 b 相加，并返回这个结果。由于它们是形式上的参数名，因此也可以换成读者自己喜欢的参数名。



注意：

由于该函数返回一个整数，因此返回类型由原来的 void 换成了 int。

【例 2.3】 两个数相加。

 实例位置：光盘\MR\Instance\02\2.3

```
#include "stdafx.h"
#include<iostream>
using namespace std;
int add(int x, int y)                         //定义一个函数 add，有两个参数 a 和 b
{
    return x+y;                               //返回两个数相加的值
}
int _tmain(int argc, _TCHAR* argv[])          //主函数
{
```






```
    int a,b,c;                          //定义变量
    cout<<"请输入两个整数：";
    cin>>a;                             //输入 a、b 的值
    cin>>b;
    c = add(a,b);                       //调用函数 add，将值赋给 c
    cout<<"a+b="<<c<<endl;              //输出 c 的值
    return 0;
}
```

运行程序，显示效果如图 2.8 所示。

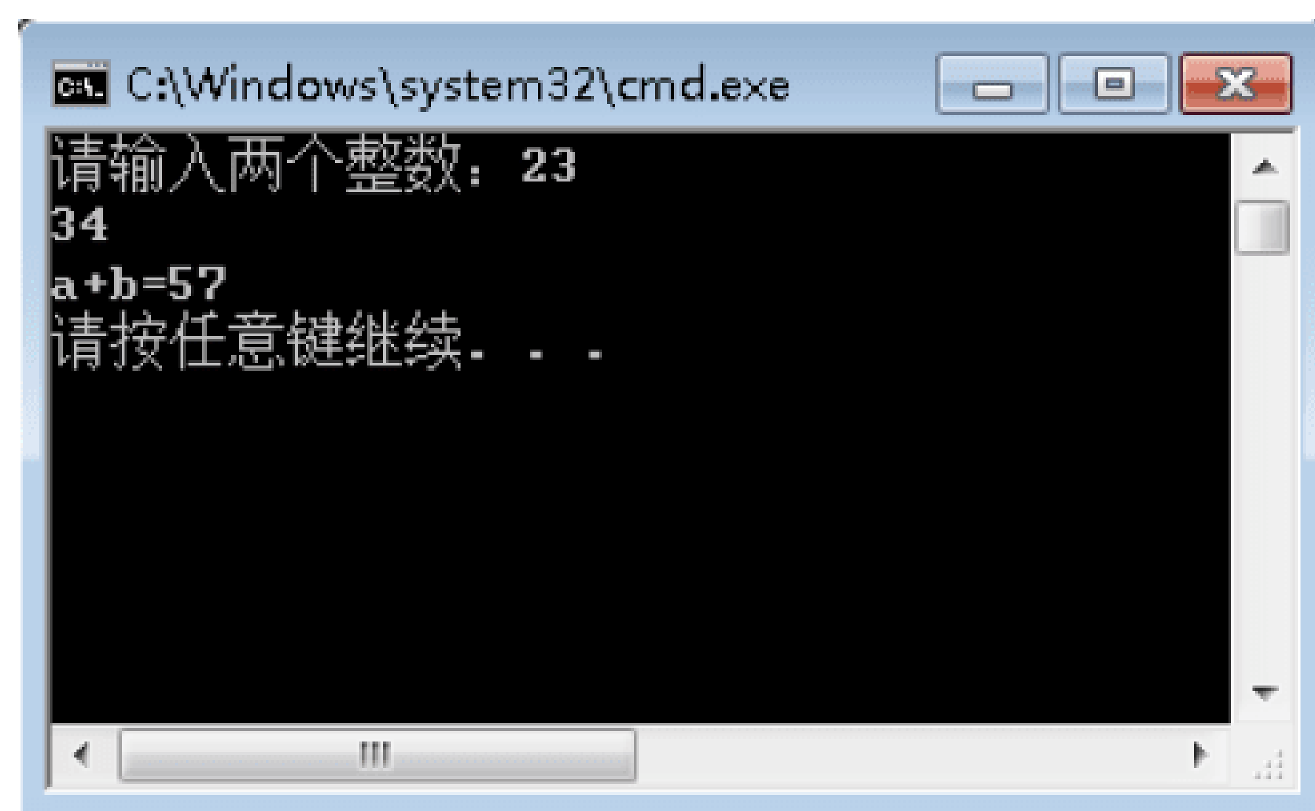


图 2.8　执行结果



注意：

cin 语句的功能与 cout 相反，cout 用来输出，而 cin 用来输入。

2.3.3　函数的返回值、参数与变量

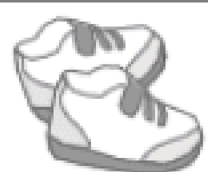
函数可以返回一个值，也可以不返回值，假如不想让函数返回值，而仅仅是执行某个功能，如输出一行文字，那么可以将函数的返回值定义为 void，如：

```
void show()                             //定义 show 函数
{
    printf("神奇的 C++之旅");            //输出语句
}
```

假如想要让函数返回一个值，如返回一个整数，那么可以将函数的返回值定义为 int，如：

```
int add(int x, int y)                   //定义一个函数 add，有两个参数 a 和 b
{
    return x+y;                         //返回两个数相加的值
}
```

关键字 return 的中文译义即返回的意思，其后的 x+y 即返回值。假如 add 函数的形式参数 x





和 y 分别传递了 23 和 34，这样 add 函数中 x 的值被初始化为 23，y 的值被初始化为 34，那么返回值便是 57。由于这个返回值是个整数，因此将 add 函数的返回值类型设置为 int，这就是函数的返回值；参数便是 add 函数小括号中的 x 和 y；而变量则是为 add 函数传递的值，如：

```
c = add(a,b);                                   //调用函数 add，将值赋给 c
```

该语句调用了 add 函数，并为该函数传递了 a 和 b，由于 a 和 b 的值不是固定不变的，可以随时改变，因此 a 和 b 叫变量，返回值用来初始化左边的 int 型变量 c，这样 c 的值便是 a 和 b 相加的和。

由于自定义的 add 函数接收两个 int 型变量，因此也可以这样调用 add 函数：

```
int x = 23,y = 34;
add(x,y);
```

第 1 行定义了两个 int 型变量 x 和 y，第 2 行调用 add 函数并为其传递这两个变量。这样，该函数返回的值即是 x+y 的值，也就是 57。

2.3.4　函数的声明和定义

想要在程序中正确地使用自定义的函数，则必须首先对其进行声明，然后再定义，声明的目的是告诉编译器即将要定义的函数的名字、返回值的类型以及参数是什么。而定义是告诉编译器这个函数的功能是什么。假如不声明，那么该函数就不能被其他函数调用。通常，我们把函数声明叫做函数原型，而把函数定义叫做函数实现。

【例 2.4】 演示函数的声明和定义。

实例位置：光盘\MR\Instance\02\2.4

```
#include "stdafx.h";
#include <iostream>
using namespace std;
int add(int x,int y);                           //函数声明
int main()
{
	int a = 5;
	int b = 6;
	cout<<add(a,b);
	return 0;
}
int add(int x,int y)                            //函数定义
{
	return x+y;
}
```

运行程序，显示效果如图 2.9 所示。





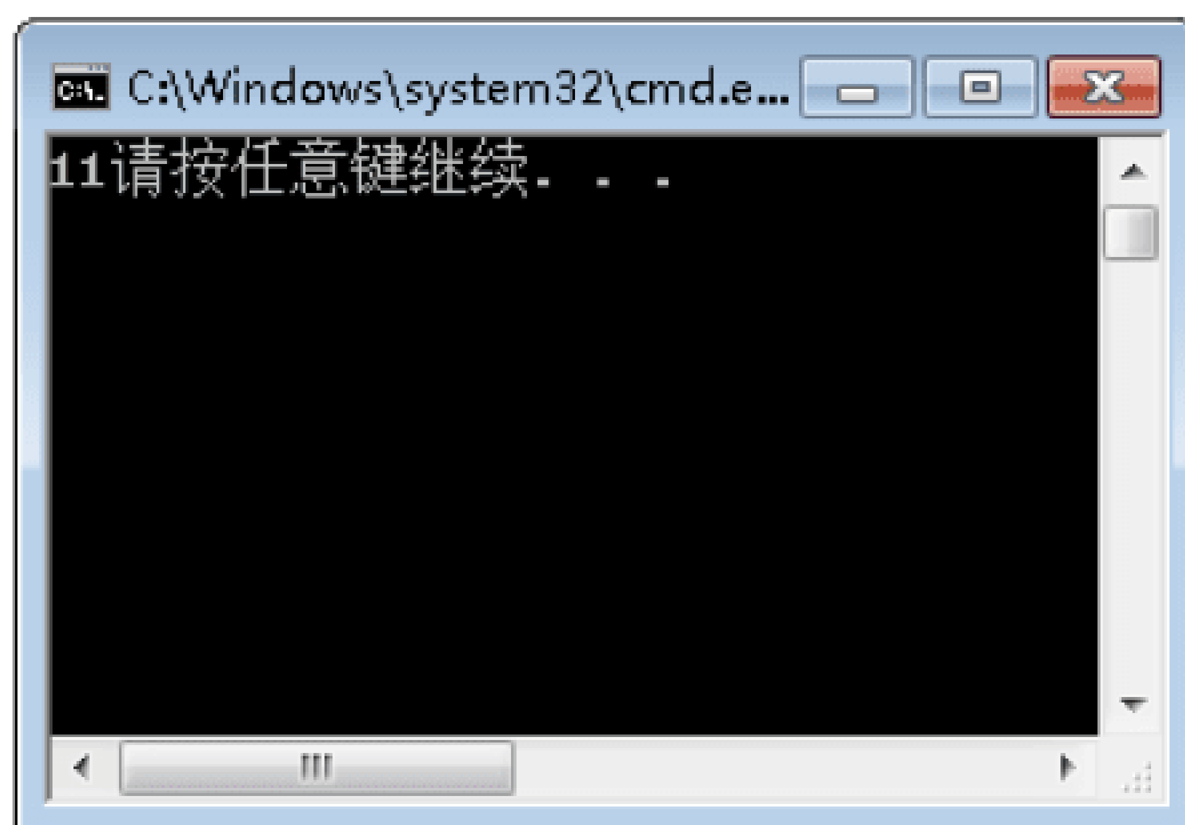


图 2.9　执行结果

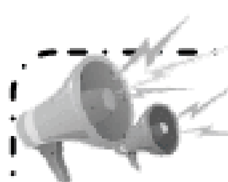


注意：

声明一个函数，仅是提供给程序员和编译器该函数的一些基本信息，它的参数名没有实际意义，因此 x 和 y 也可以省略不写，如：

```
int add(int,int)
```

这样当程序员在阅读该函数的声明时，便可了解到该函数的一些初步信息，如该函数返回一个整数，并接收两个整数或者整型变量。



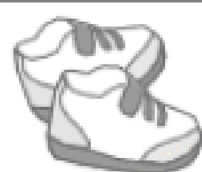
注意：

声明和定义的区别为声明只是告诉编译器将要有这样的一个函数，在内存中它并不为这个函数分配内存，而只有在定义的时候才会给这个函数分配内存。

许多时候也可以不用声明，直接定义一个函数，如实例 2.4 写成下面的形式也可编译成功：

```
#include "stdafx.h";
#include <iostream>
using namespace std;
int main()
{
	int a = 5;
	int b = 6;
	cout<<add(a,b);
	return 0;
}
int add(int x,int y)                                    //函数定义
{
	return x+y;
}
```

虽然这样可以编译成功，但这不是一个良好的编程习惯。在某些情况下，也会编译失败，例如下面的例子。





Note

【例 2.5】 函数无声明。

实例位置：光盘\MR\Instance\02\2.5

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void A()
{
	cout<<"函数 A\n";
	B();                                        //调用 B 函数
}
void B()
{
	cout<<"函数 B\n";
	A();                                        //调用 A 函数
}
int main()
{
	A();
	B();
	return 0;
}
```

此实例中，函数 A 要调用函数 B，而函数 B 也要调用函数 A。由于无法既使函数 A 在函数 B 之前定义，又使函数 B 在函数 A 之前定义，因此该程序在执行后会报错，如图 2.10 所示。

错误列表

1 个错误 | 0 个警告 | 0 个消息

	说明	文	行	列	项目
1	error C3861: "B"：找不到标识符	函数	10	1	函数无声明实例

图 2.10 错误列表

函数 B 没有声明，即编译器并不知道有这个函数，因此需要先声明这个函数 B。同时编译器也不知道 A 这个函数，因此也需要声明函数 A，例如下面的例子。

【例 2.6】 函数有声明。

实例位置：光盘\MR\Instance\02\2.6

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void A();
```






```
void B();                                   //函数 A、B 声明
void A()
{
	cout<<"函数 A\n";
	B();                                    //调用 B 函数
}
void B()
{
	cout<<"函数 B\n";
	A();                                    //调用 A 函数
}

int main()
{
	A();
	B();
	return 0;
}
```

在声明了函数 A 和 B 之后，便没有错误了。



注意：

由于函数 A 中调用了函数 B，而函数 B 又调用了函数 A，因此，该程序运行后会产生一个无穷循环。

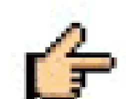
2.3.5　变量

在 C++语言中，声明的变量主要分为全局变量和局部变量，其可以出现在程序的任何位置，在不同的位置声明，其作用域不同。

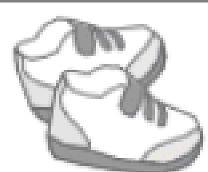
- ☑ 全局变量：其说明语句不在任何一个类定义、函数定义或复合语句（程序块）中的变量。全局变量所占用的控件在内存的数据区，在程序运行的整个过程中位置保持不变。
- ☑ 局部变量：其说明语句在某一个类定义、函数定义或复合语句（程序块）中的变量。局部变量所占用的空间在为程序运行时所设置的临时工作区中，以堆栈的形式允许反复占用和释放。

下面用一个实例来说明局部变量具体的作用域。

【例 2.7】 局部变量。

 **实例位置：光盘\MR\Instance\02\2.7**

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()                                  //主函数
```





Note

```
{
    int a = 0;                                   //定义全局变量 a，b
    int b = 0;
    a++;                                         //a，b 累加
    b++;
    cout<<"a="<<a<<","<<"b="<<b<<endl;           //输出 a，b 的值
    {
        float a = 0.5;                           //定义局部变量 a
        a++;                                     //a，b 累加
        b++;
        cout<<"a="<<a<<","<<"b="<<b<<endl;       //输出 a，b 的值
    }
    a++;                                         //a，b 累加
    b++;
    cout<<"a="<<a<<","<<"b="<<b<<endl;           //输出 a，b 的值
}
```

运行程序，显示效果如图 2.11 所示。

```
C:\Windows\system32\cmd.exe
a=1,b=1
a=1.5,b=2
a=2,b=3
请按任意键继续. . .
```

图 2.11　执行结果



注意：

在实际的应用程序中，如果涉及全局变量，读者应仔细分析其中每个变量在程序中的作用范围及判断其值的变化。

2.4　C++语言基本要素

程序设计语言的基本要素包括标识符、关键字、常量和变量等。本节将介绍 C++语言的基本要素。





2.4.1　解读标识符

在 C++语言中，变量、常量、函数、标签和用户定义的各种对象，被称为标识符。标识符由一个或多个字符构成。字符可以是字母、数字或下划线，但是标识符的首字符必须是字母或下划线，而不能是数字。例如，下面的标识符均是合法的：

```
maxAge, num,_sex
```

而下面的标识符是非法的：

```
1maxAge,nu!m,
```

在 C++语言中，标识符是区分大小写的。例如，value 和 Value 是两个不同的标识符。此外，标识符不能与 C/C++的关键字同名。



注意：

C++语言中标识符的长度可以是任意的，但是通常情况下，前 1024 个字符是有意义的，这与 C 语言不同。在 C 语言中，标识符也可以是任意长度，但是在外部链接进程中调用该标识符时，通常前 6 个字符是有效的，如被多个文件共享的全局函数或变量。如果标识符不用于外部进程链接，通常前 31 个字符是有效的。

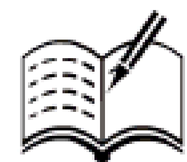
2.4.2　关键字

关键字是 C++编译器内置的有特殊意义的标识符，用户不能定义与关键字相同的标识符。C++语言关键字如表 2.1 所示。

表 2.1　C++关键字

__asm	class	double	float
__assume	const	dynamic_cast	for
auto	const_cast	else	friend
__based	continue	enum	goto
bool	__declspec	__except	if
break	default	explicit	inline
case	delete	extern	__inline
catch	dllexport	false	int
__cdecl	dllimport	__fastcall	__int8
char	do	__finally	__int16





Note

续表

__int32	noreturn	static_cast	typeid
__int64	operator	__stdcall	typename
__leave	private	struct	union
long	protected	switch	unsigned
main	public	template	using declaration，using directive
__multiple_inheritance	register	this	uuid
__single_inheritance	reinterpret_cast	thread	__uuidof
__virtual_inheritance	return	throw	virtual
mutable	short	true	void
naked	signed	try	volatile
namespace	sizeof	__try	wmain
new	static	typedef	while

2.4.3　定义和使用常量

常量，顾名思义，其值在运行时是不能改变的，但是在定义常量时可以设置初始值。在 C++中，可以使用 const 关键字来定义一个只读变量。用 const 修饰的变量不能够用赋值、自增自减等运算改变其值。例如，下面的代码编译会出错：

```
const int MAX_VALUE;                    //声明一个常量
MAX_VALUE = 100;                        //给常量赋值，错误
```

正确的代码如下：

```
const int MAX_VALUE = 100;              //初始化一个变量，正确
```

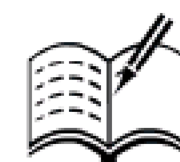
对于常量，编译器会将其放置在一个只读的内存区域（文字常量区），其值不能被修改，但是可以应用在各种表达式中。如果用户试图修改常量，编译器将提示错误。

使用常量的最大好处是灵活。当程序中有多处需要使用一个常数值时，可以使用常量代替。当需要改动常数值时，只需要改动常量的值即可。此外，在定义函数时，如果在函数体中不需要修改参数值，建议将参数的类型定义为常量，这样当用户不小心在函数体内修改了参数值时，编译器将提示错误信息。

2.4.4　变量的应用

其值可以改变的量称为“变量”。变量提供了一个具有名称（变量名）的存储区域，使得开发人员可以通过名称来对存储区域进行读写。与常量不同的是，变量可以在程序中被随意赋值。





对于每一个变量，都具有两个属性，也就是通常所说的左值和右值。所谓左值，是指变量的地址值，即存储变量值的内存地址。右值是指变量的数据值，即内存地址中存储的数据。

在程序中定义变量时，首先是变量的数据类型，然后是变量名。如下面的代码定义了两个变量：

```
int min = 0 ;                    //定义一个整型变量
char* pch ;                      //定义一个指针变量
```

在定义变量时，可以对变量进行初始化，即为其设置初始值。例如，上面的代码定义了一个 min 整型变量，将其初始化为 0。在初始化变量时，可以将变量初始化为自身。例如：

```
int min = min;
```

这样做虽然是合法的，但也是很愚蠢和不明智的。在初始化变量时，可以进行隐式初始化。例如：

```
int min(10);                     //初始化为 10，等同"int min = 10;"
```

当一条语句定义多个变量时，可以为多个变量同时指定初始值，并且后续变量可以利用之前变量作为初始值。例如：

```
int min = 10 , max = min + 50;
```



说明：

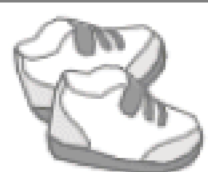
在用一条语句定义多个变量时，变量之间用逗号分隔，在最后一个变量定义结束后，以分号结束语句。

2.5　C++代码编写规范

C++程序语言的书写格式自由度高，灵活性强，随意性大。如一行内可写一条语句，也可写几条语句；一个语句也可分写在多行内，从而使得 C++程序比其他语言更难理解。为了提高程序的可读性，使用规范的代码编写是非常重要和必要的。

2.5.1　代码写规范的好处

代码写规范可以使程序结构清晰、明了，程序代码紧凑，增加了程序的可读性，特别是在团队中开发程序。因此，写代码时遵守 C++的规范是非常重要的。





Note

2.5.2 如何将代码写规范

为了增加程序的可读性和便于理解，编写程序时应按以下要点书写：

（1）一般情况下每个语句占用一行。

（2）表示结构层次的大括弧，写在该结构化语句第 1 个字母的下方，与结构化语句对齐，并占用一行。例如：

```
void main()
{
    cout<<"我会学好 C++!! "<<endl;
}
```

（3）不同结构层次的语句，从不同的起始位置开始，即同一结构层次中的语句缩进同样的字数。例如：

```
{
…
if(i<0)
j = -i;                    //如果 i 是负数，j 的值为 i 的相反数
else
j = i;                     //如果 i 不是负数，j 的值为 i 的值
…
}
```

2.6 综 合 应 用

【例 2.8】 使用输出语句输出一个正方形，输出结果如图 2.12 所示。

程序设计步骤如下：

（1）新建控制台应用程序。

（2）引用头文件。

（3）运用 printf 语句将图像输出。

（4）程序主要代码如下。

实例位置：光盘\MR\Instance\02\2.8

```
#include "stdafx.h"
void main()
{
    printf("* * *    *\n");
```



```
        printf("*          *\n");
        printf("*          *\n");
        printf("*          *\n");
        printf("* * *      *\n");
}
```



Note

运行程序，显示效果如图 2.11 所示。

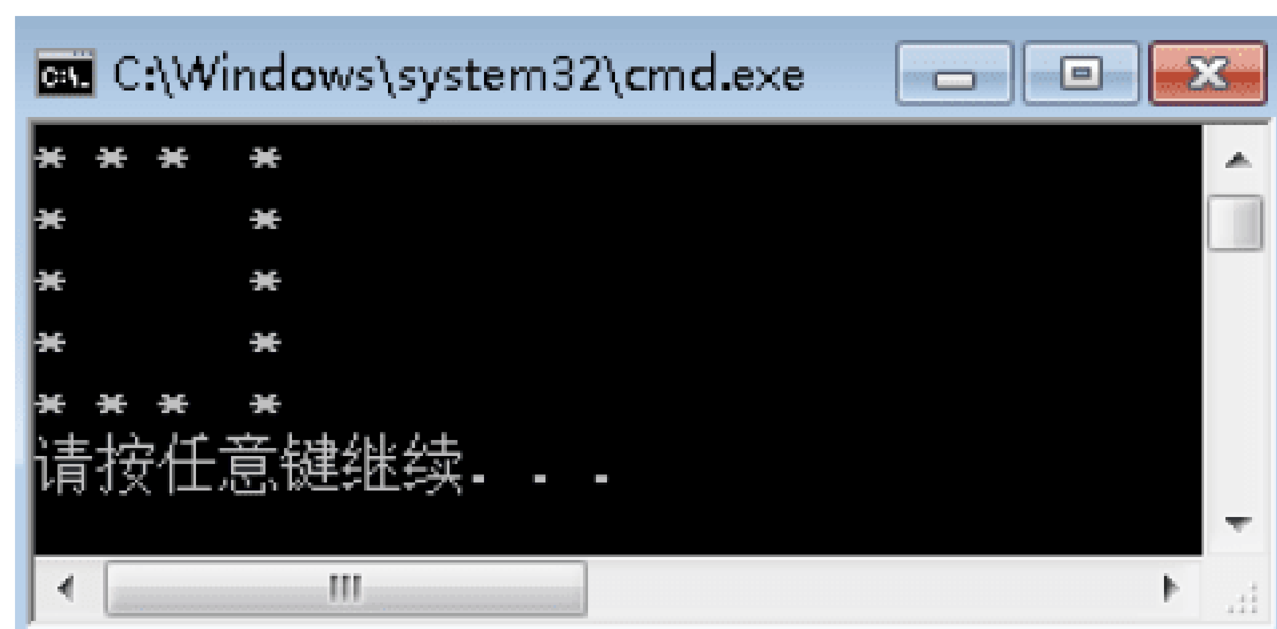


图 2.12　执行结果

2.7　本章常见错误

如果函数名前是 void，函数体最后不用写 return。

void 表示函数的返回值，函数的返回值是用来判断函数执行情况以及返回函数执行结果的。void 代表不返回任何数据。如果要返回还需要使用 return 语句。

2.8　本 章 小 结

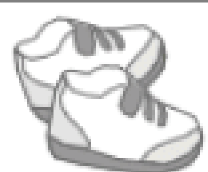
本章分为两个部分，第 1 部分编写了一个简短的 C++程序，通过这个程序讲解了 C++的基本组成：它是由一个预处理标志、一个预处理指令、一个头文件和一个 main 函数组成，演示了注释的作用及如何添加注释；第 2 部分讲解了 C++中使用最频繁的单元——函数。函数是完成一定功能的代码块，当程序的某个地方要执行该程序时，只需要输入函数名即可。

2.9　跟 我 上 机

参考答案：光盘\MR\跟我上机

在 C++中编写一个控制台应用程序，要求实现以下功能：

（1）提示用户输入一个数值。



（2）根据输入的数值，计算并输出它的绝对值。



Note

```
#include "stdafx.h"
#include<iostream>
using namespace std;
int main()
{
	int a,b;                                   //定义两个变量
	cout<<"请输入一个数：\n";
	cin>>a;                                    //输入一个数
	if(a>=0)                                   //如果这个数大于等于 0
		b = a;                                 //b 等于这个数
	else                                       //如果这个数小于 0
		b = -a;                                //b 等于这个数的负数
	cout<<b;                                   //输出 b
	return b;
}
```

第3章

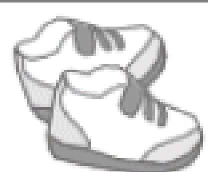
变量和数据类型

（视频讲解：1 小时 5 分钟）

数据类型是C++语言的基础，要学习一门编程语言首先要掌握它的数据类型，不同的数据类型占用不同的内存空间，合理定义数据类型可以优化程序的运行。本章将介绍变量及数据类型。

本章能够完成的主要范例（已掌握的在方框中打勾）

- [ ] 掌握 C++中的常量、变量及其定义
- [ ] 掌握数据类型的分类
- [ ] 了解数据的输入与输出





3.1 常 量

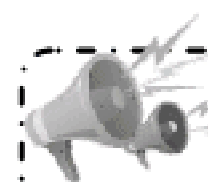
C++程序中的数据可分为常量与变量两大类。常量是在程序运行过程中不变的量，变量是在程序运行过程中可发生变化的值。常量可分为整型常量、实型常量、字符常量和字符串常量 4 种。本节主要针对常量进行分类介绍。

3.1.1 整型常量

整型常量分为有符号整型常量和无符号整型常量两种类型。

-225 代表一个负数，+1024 代表一个正整数，正整数前面的“+”符号可以省略，即+1024 和 1024 表示的意义相同。

由于基本的数据类型中除了整型外，还有长整型和短整型，所以整型常量也有长整型常量和短整型常量之分。长整型常量不是可以无限大的，它的最大值是有限的，根据 CPU 寄存器位数的不同以及编译器的不同，最大的整型常量值也会不同。



注意：

4294967295 是 32 位 CPU 寄存器以及 VC6 编译器所允许的最大正整数。

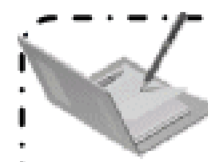
整型常量在编写代码时不仅可以写成十进制整数形式，也可以写成十六进制或八进制整数形式。

（1）八进制形式整型常量必须以 0 开头，即以 0 作为八进制数的前缀，每位取值范围是 0～7。八进制数通常是无符号数。

- ☑ 合法的八进制数：016、0101、0127。
- ☑ 不合法的八进制数：256 无前缀 0，它代表十进制整型常量；0396 中数字 9 不是八进制应有的取值。

（2）十六进制整型常量的前缀是 0X 或 0x，其数码取值范围为 0～9，以及 A～F 或 a～f。

- ☑ 合法的十六进制整常数：0X2A1、0XC5、0XFFFF。
- ☑ 不合法的十六进制整常数：5A 无前缀 0X；0X3N 中含有非十六进制数 N。



说明：

合法是指能通过编译器编译，非法则是不能通过编译器编译。

3.1.2 实型常量

实型常量也称为浮点数，只能采用十进制形式表示。它有两种表示形式，即小数表示法和指





数表示法。

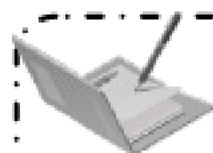
1. 小数表示法

使用这种表示法，实型常量由整数部分和小数部分组成，整数部分和小数部分每位取值范围是 0～9，中间用小数点分隔。例如，0.0、3.25、0.00666、4.0、-4.1、-0.0001 等均为合法的实型常量。

整数部分和小数部分有时可以不必同时出现，例如.2 和 2.。

2. 指数表示法

指数表示法也称科学计数法，指数部分以符号“e”或“E”开始，但必须是整数，并且符号“e”或“E”两边必须有一个数，例如 1.2e20 和-3.4e-2。



说明：

在字母 e（或 E）之前的小数部分中，小数点左边应有一位（且只能有一个）非零的数字，称为规范化的指数形式。

科学计数法中 23e3 表示 23 乘以 10 的 3 次方。

L 代表长整型。L 可以是大写或小写，在编写代码时也可以不写。U 和 u 代表无符号，例如，255U 或 255u 都代表无符号整型常量 255。

符号 L 或 l 与符号 U 或 u 可以一起使用。65536lu 代表无符号长整型常量 65536。

C++编译系统把用这种形式表示的浮点数按双精度常量处理，在内存中占 8 个字节。如果在实数的数字之后加字母 F 或 f，表示此数为单精度浮点数，如 123F 或-43f 占 4 个字节。如果加字母 L 或 l，表示此数为长双精度数（long double）。

3.1.3　字符常量

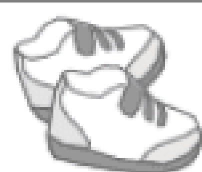
字符常量是用单引号括起来的一个字符，例如‘a’和‘?’都是合法的字符常量。在对代码编译时，编译器会根据 ASCII 码表将字符常量转换成整数常量。字符‘a’的 ASCII 码值是 97，字符‘A’的 ASCII 码值是 65，字符‘?’的 ASCII 码值是 63。ASCII 码表中还有很多通过键盘无法输入的字符，可以使用‘\ddd’或‘\xhh’来引用这些字符。可以使用‘\ddd’或‘\xhh’来引用 ASCII 码表中的所有字符。ddd 是 1～3 位八进制数所代表的字符，\xhh 是 1～2 位十六进制数所代表的字符。例如，‘\101’表示 ASCII 码“A”，‘\XOA’代表换行等。

【例 3.1】 转义字符的实例。



实例位置：光盘\MR\Instance\03\3.1

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
{
```



```
    cout<<"A"<<endl;
    cout<<"\101"<<endl;
    cout<<"\x41"<<endl;
    cout<<"\052.\x1E"<<endl;
}
```



Note

运行程序，显示效果如图 3.1 所示。

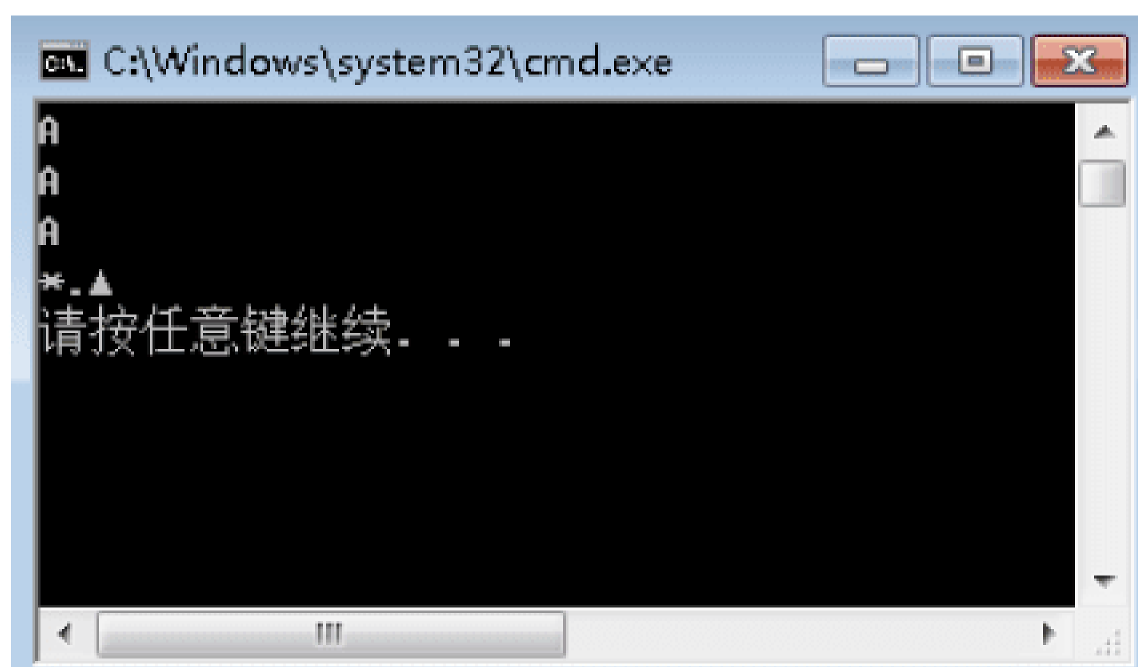


图 3.1　执行结果

转义字符是特殊的字符常量，使用时以字符“\”代表开始转义，与后面不同的字符组合表示转义后的字符。转义字符如表 3.1 所示。

表 3.1　转义字符说明

转 义 字 符	意　　义	ASCII 代码
\0	空字符	0
\n	换行	10
\t	水平制表	9
\b	退格	8
\r	回车	13
\f	换页	12
\\	反斜杠	93
\,	单引号字符	39
\”	双引号字符	34
\a	响铃	7

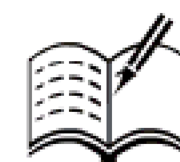
3.1.4　字符串常量

字符串常量是由一对双引号括起来的零个或多个字符序列，如“welcome to my home”、“hello C++”。“ ”表示一个空字符串。

同样，‘ ’也可以表示空字符，而 NULL 是一种特殊的数据类型，表示空的意思。有的编译器把它编译成 0，有的则编译成其他值。

字符串常量实际上是一个字符数组，可以将字符串分解成若干个字符，字符的数量是字符串的长度。字符串常量一般都是用来给字符数组变量赋值或是直接作为实参传递，为告知编译器字符串已经结束。一般在给字符数组赋初值时在字符串的末尾加上字符‘\0’，表示字符结束，如





果不加字符结束标志，有可能会出现意想不到的错误。

字符常量‘A’与字符串常量“A”是不同的，字符串常量“A”是由‘A’和‘\0’两个字符组成的，字符串的长度是 2，字符常量‘A’只是一个字符，没有长度。

3.1.5　其他常量

前面讲到的都是普通的常量，常量还包括布尔常量、枚举常量、宏定义常量等。

☑　布尔（bool）常量：只有两个，一个是 true，表示真；一个是 false，表示假。

☑　枚举常量：枚举型数据中定义的成员也都是常量，这将在后面介绍。

☑　宏定义常量：通过#define 宏定义的一些值也是常量。例如：

```
#define PI 3.1415;
```

其中 PI 就是常量。

3.2　变　　量

前面已经讲解了常量，即值不能改变量，那么值可以改变的量就叫做变量。本节将介绍变量的相关知识。

3.2.1　标识符

标识符（identifier）是用来对 C++程序中的常量、变量、语句标号以及用户自定义函数的名称进行标识的符号。

标识符命名规则如下：

（1）由字母、数字及下划线组成，且不能以数字开头。

（2）大写和小写字母代表不同意义。

（3）不能与关键字同名。

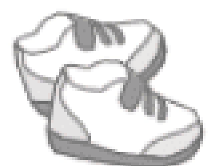
（4）尽量“见名知意”，应该受一定规范的约束。

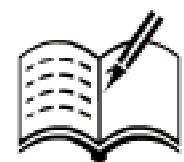
如 6A、ABC*（不能使用*）、case（是保留字）都是不合法的标识符。

C++有许多保留关键字，如表 3.2 所示。

表 3.2　C++保留关键字

asm	auto	break	case	catch	char	class	const	continue
default	selete	do	double	else	enum	extern	float	for
friend	goto	if	inline	int	long	new	operator	overload
private	protected	public	register	teturn	short	signed	sizeof	static
struct	switch	this	template	throw	try	typedef	union	unsigned
virtual	void	volatile	while					





Note

3.2.2 变量与变量说明

变量是指程序在运行时其值可改变的量。每个变量都由一个变量名标识，又具有一个特定的数据类型。

变量在使用之前一定要定义或说明，声明的一般形式如下：

```
[修饰符] 类型 变量名标识符
```

类型是变量类型的说明符，说明变量的数据类型。修饰符是任选的，可以没有。

多个同一类型的变量可以在一行中声明，不同变量名用逗号运算符隔开。例如：

```
int a,b,c;
```

与

```
int a;
int b;
int c;
```

两者等价。

3.2.3 整型变量

整型变量可以分为短整型、整型和长整型，变量类型说明符分别是 short、int、long。根据是否有符号还可以分为以下 6 种：

```
整型            [signed] int
无符号整型      unsigned [int]
有符号短整型    [signed] short [int]
无符号短整型    unsigned short [int]
有符号长整型    [signed] long [int]
无符号长整型    unsigned long [int]
```

方括号中的关键字表示可以省略，例如，[signed] int 可以写成 int。

短整型 short 在内存中占用两个字节的空间，可以表示的数值范围是-32768～32767，如果是无符号短整型 unsigned short，表示的数值范围是 0～65535。整型 int 占用 4 个字节的空间，有符号整型表示的数值范围是-2147483648～2147483648，无符号整型 unsigned int 表示的数值范围是 0～4294967295。长整型与整型占用的字节数相同，表示的数值范围也相同，具体如表 3.3 所示。

表 3.3 整数类型

类　型	名　称	字　节　数	范　围
[signed] int	有符号整型	4	-2147483648～2147483647
Unsigned [int]	无符号整型	4	0～4294967295



续表

类　型	名　称	字 节 数	范　围
[signed]short	有符号短整型	2	-32768～32767
Unsigned short [int]	无符号短整型	2	0～65535
[signed] long [int]	有符号长整型	4	-2147483648～2147483647
Unsigned long [int]	无符号长整型	4	0～4294967295



Note

3.2.4　实型变量

实型变量又可称为浮点型变量，可分为单精度（float）、双精度（double）和长双精度（long double）3 种。

Visual C++ 6.0 中，对 float 提供 6 位有效数字，对 double 提供 15 位有效数字，并且 float 和 double 的数值范围不同。对 float 分配 4 个字节，对 double 和 long double 分配 8 个字节。

1．单精度

类型说明符为 float，该实型数据在内存中占 4 个字节，表示的数值范围是-3.4e38～3.4e38。例如：

```
float a;
```

2．双精度

类型说明符为 double，该实型数据在内存中占 8 个字节，表示的数值范围是-1.7e308～1.7e308。例如：

```
double b;
```

3．双长精度

类型说明符为 long double，该实型数据在内存中占 10 个字节，表示的数值范围是-1.1e4932～1.1e4932。例如：

```
long double c;
```

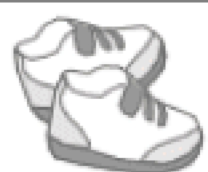
3.2.5　变量赋值

变量值是动态改变的，每次改变都需要进行赋值运算。变量赋值的形式如下：

```
变量名标识符 = 表达式;
```

变量名标识符就是声明变量时定义的，表达式将在后面的章节中讲到。例如：

```
int i;                          //声明变量
i = 100;                        //给变量赋值
```



声明 i 是一个整型变量，100 是一个常量。

```
inti,j;                    //声明变量
i = 100;                   //给变量赋值
j = i;                     //将一个变量的值赋给另一个变量
```



3.2.6　变量赋初值

可以在声明变量时就把数值赋给变量，这个过程叫做变量赋初值，赋初值的情况有以下几种：

```
int x = 5;
```

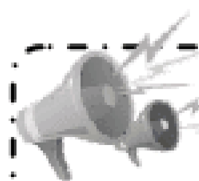
表示定义 x 为有符号的基本整型变量，赋初值为 5。

```
int x,y,z = 6;
```

表示定义 x、y、z 为有符号的基本整型变量，z 赋初值为 6。

```
int x = 2,y = 2,z = 2;
```

表示定义 x、y、z 为有符号的基本整型变量，且赋予的初值均为 2。



注意：

定义变量并赋初值时可以写成“int x = 3, y = 3,z = 3;”，但不可写成“int a = b = c = 3;”。

3.2.7　字符变量

字符变量的类型说明符为 char，一个字符变量占用 1 字节内存单元。例如，“char ch1;”用于定义一个字符变量 ch1：“ch1 = ‘a’;”用于给字符变量赋值。

字符变量值在内存中存储的是字符的 ASCII 码，即一个无符号整数，其形式与整型变量的存储形式一样，字符型数据与整型数据之间通用。

（1）一个字符型数据，既可以字符形式输出，也可以整数形式输出。

【例 3.2】 字符型数据与整型数据间的运算。



实例位置：光盘\MR\Instance\03\3.2

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	char c1,c2;
	c1 = 'a';
	c2 = 'b';
```



```
	int i1,i2;
	i1 = 'a';
	i2 = 'b';
	printf("%c,%d\n%c,%d",c1,i1,c2,i2);
}
```

运行程序，显示效果如图 3.2 所示。

（2）允许对字符数据进行算术运算，此时就是对它们的 ASCII 码值进行算术运算。

【例 3.3】 字符型数据进行算术运算。

实例位置：光盘\MR\Instance\03\3.3

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	char ch1,ch2;
	ch1 = 'a';ch2 = 'B';                          //给 ch1，ch2 赋值
	printf("ch1 = %c,ch2 = %c\n",ch1-32,ch2+32);  //用字符形式输出一个大于 256 的数值
	printf("ch1+10 = %d\n",ch1+10);
	printf("ch1+10 = %c\n",ch1+10);
	printf("ch2+10 = %d\n",ch2+10);
	printf("ch2+10 = %c\n",ch2+10);
}
```

运行程序，显示效果如图 3.3 所示。

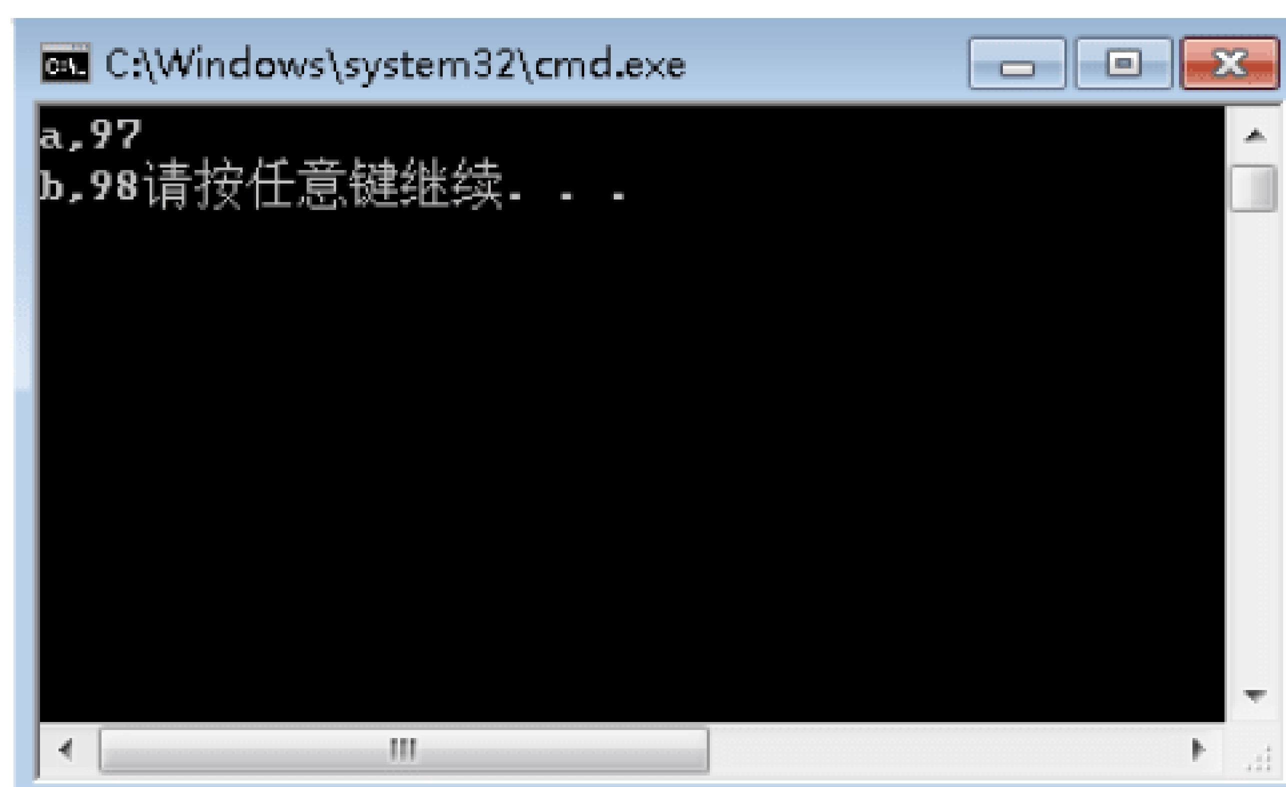


图 3.2　执行结果

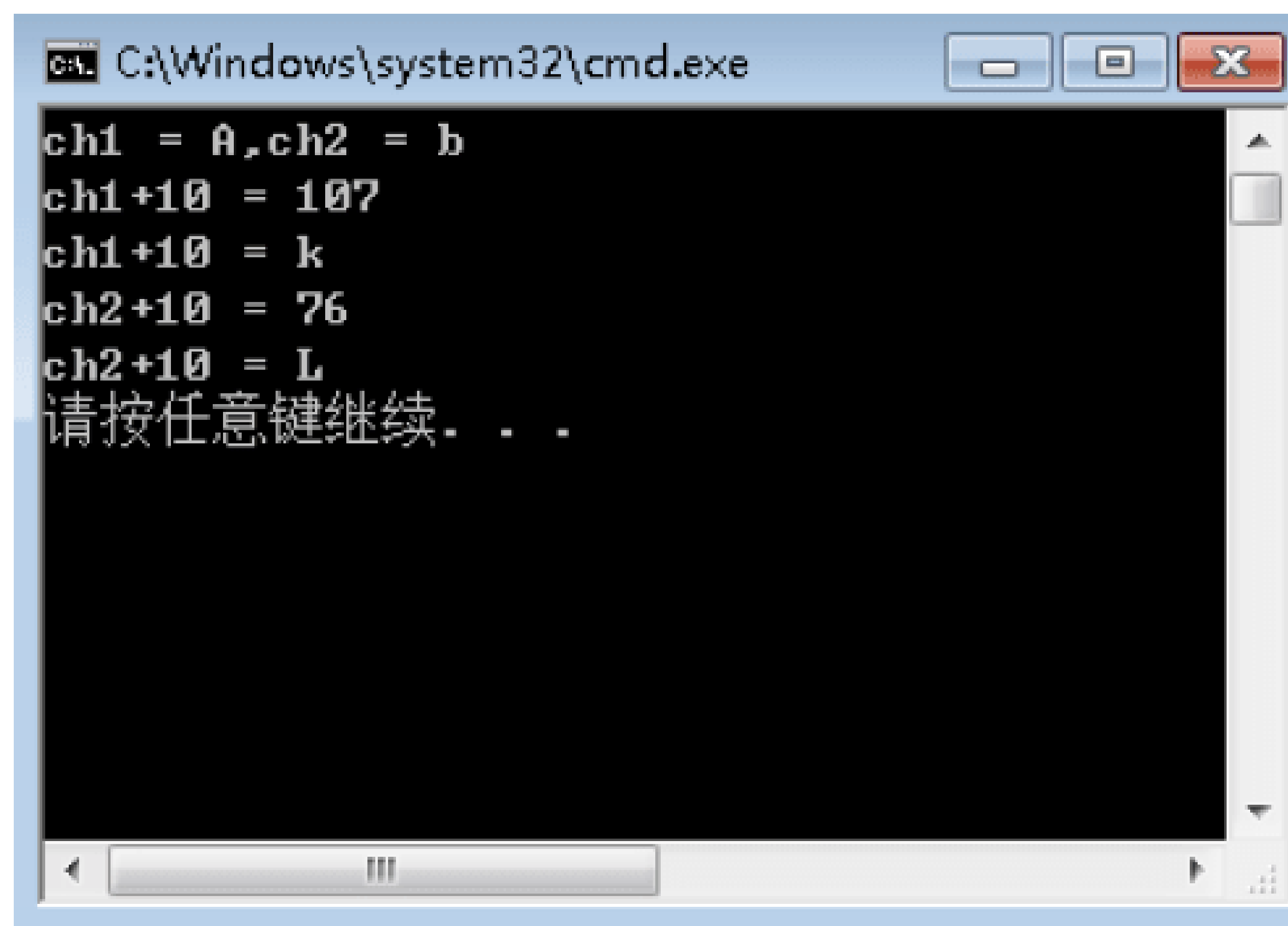
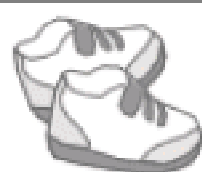


图 3.3　执行结果

3.3　常用数据类型

C++语言中常用的数据类型有数值类型、字符类型、数组类型、布尔类型、枚举类型、结构体类型、共用体类型、指针类型、引用类型和自定义类型。本节将详细介绍这些数据类型。



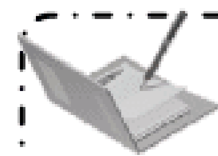
3.3.1　定义数值类型



C++语言中数值类型主要分为整型和实型（浮点类型）两大类。其中，整型按符号划分，可以分为有符号和无符号两大类；按长度划分，可以分为普通整型、短整型和长整型 3 类，如表 3.4 所示。

表 3.4　整数类型

类　型	名　称	字 节 数	范　围
[signed] int	有符号整型	4	−2147483648～2147483647
Unsigned [int]	无符号整型	4	0～4294967295
[signed]short	有符号短整型	2	−32768～32767
Unsigned short [int]	无符号短整型	2	0～65535
[signed] long [int]	有符号长整型	4	−2147483648～2147483647
Unsigned long [int]	无符号长整型	4	0～4294967295



说明：

表格中的[]为可选部分。例如，[signed] long [int]可以简写为 long。

实型主要包括单精度型、双精度型和长双精度型，如表 3.5 所示。

表 3.5　实数类型

类　型	名　称	字 节 数	范　围
float	单精度型	4	$1.2e^{-38}$～$3.4e^{38}$
double	双精度型	8	$2.2e^{-308}$～$1.8e^{308}$
long double	长双精度型	8	$2.2e^{-308}$～$1.8e^{308}$

在程序中使用实型数据时需要注意以下几点。

（1）实数的相加

实型数据的有效数字是有限制的，如单精度 float 的有效数字是 6～7 位，如果将数字 86041238.78 赋值给 float 类型，显示的数字可能是 86041240.00，个位数 8 被四舍五入，小数位被忽略。如果将 86041238.78 与 5 相加，输出的结果为 86041245.00，而不是 86041243.78。

（2）实数与零的比较

在开发程序的过程中，经常会进行两个实数的比较，此时尽量不要使用“==”或“!=”运算符，而应使用“>=”或“<=”之类的运算符，许多程序开发人员在此经常犯错。如：

```
float fvar = 0.00001;                    //定义一个浮点型变量
if (fvar == 0.0)                         //判断是否为 0
…
```

上述代码并不是高质量的代码，如果程序要求的精度非常高，可能会产生未知的结果。通常





在比较实数时需要定义实数的精度。

【例 3.4】 利用实数精度进行实数比较。

实例位置：光盘\MR\Instance\03\3.4

```
#include “stdafx.h”
void main()
{
	float eps = 0.0000001;			//定义 0 的精度
	float fvar = 0.00001;
	if (fvar >= -eps && fvar <= eps)	//超出精度
		printf("等于零!\n",fvar);
	else
		printf("不等于零!\n",10);
}
```

执行结果如图 3.4 所示。

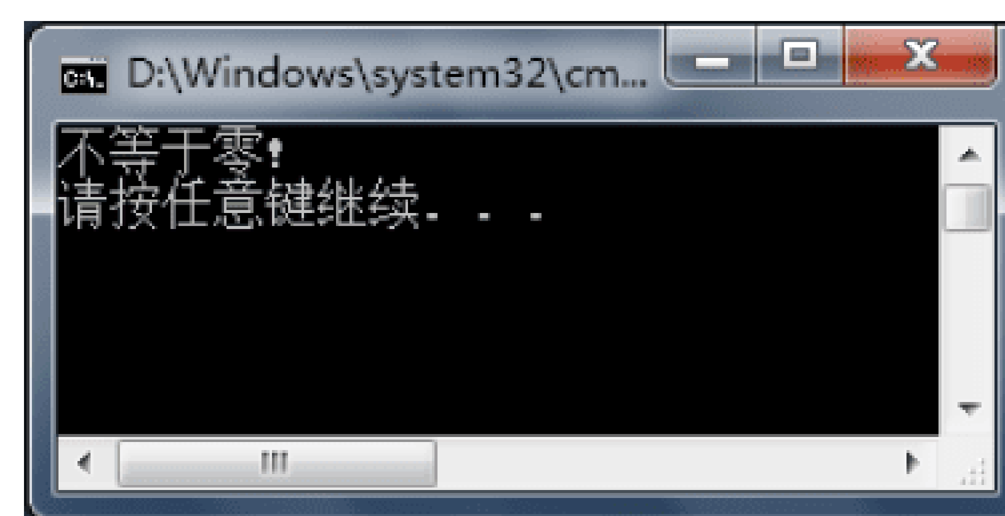


图 3.4　执行结果

3.3.2　字符类型

在 C++语言中，字符数据使用“‘’”来表示，如‘A’、‘B’、‘C’等。定义字符变量可以使用 char 关键字。例如：

```
char c= 'a';				//定义一个字符型变量
char ch = 'b';
```

在计算机中字符是以 ASCII 码的形式存储的，因此可以直接将整数赋值给字符变量。例如：

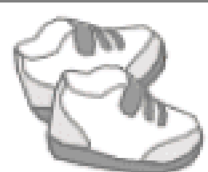
```
char ch = 97;				//同 ch = 'a';
printf("%c\n",ch);			//输出字符 a
```

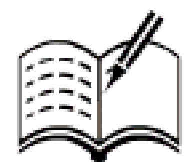
输出结果为“a”，因为 97 对应的 ASCII 码为“a”。

3.3.3　布尔类型

在逻辑判断中，结果通常只有真和假两个值。C++语言中提供了布尔类型 bool 来描述真和假。bool 类型共有两个取值，分别为 true 和 false。顾名思义，true 表示真，false 表示假。在程序中，bool 类型被作为整数类型对待，false 表示 0，true 表示 1。将 bool 类型赋值给整型是合法的，反之，将整型赋值给 bool 类型也是合法的。例如：

```
bool ret;				//定义布尔型变量
int  var = 3;
ret = var;				//ret=true
var = ret;				//var=1
```





Note

3.4 输入与输出数据

在用户与计算机进行交互的过程中，数据输入和数据输出是必不可少的操作过程，计算机需要通过输入获取来自用户的操作指令，并通过输出来显示操作结果。本节将介绍数据输入与输出的相关内容。

3.4.1 通过 printf 格式输出数据

C++语言中还保留着 C 语言中的屏幕输出函数 printf。使用 printf 可以将任意数量类型的数据输入到屏幕。

printf 函数的作用是向终端（输出设备）输出若干任意类型的数据。printf 函数的一般格式为：

```
printf(格式控制, 输出列表);
```

括号内包括两部分：

☑ 格式控制

格式控制是用双引号括起来的字符串，此处也称为转换控制字符串。其中包括两种字符，一种是格式字符，另一种是普通字符。

- 格式字符用来进行格式说明，作用是将输出的数据转换为指定的格式输出。格式字符是以“%”字符开头的。
- 普通字符是需要原样输出的字符，其中包括双引号内的逗号、空格和换行符。

☑ 输出列表

输出列表中列出的是要进行输出的一些数据，可以是变量或表达式。

例如，要输出一个整型变量时：

```
int iInt=10;
printf("this is %d",iInt);                    //输出 iInt
```

执行上面的语句显示出来的字符是 this is 10。在格式控制双引号中的字符是 this is %d，其中的 this is 字符串是普通字符，而%d 是格式字符，表示输出的是后面 iInt 的数据。

由于 printf 是函数，“格式控制”和“输出列表”这两个位置都是函数的参数，因此 printf 函数的一般形式也可以表示为：

```
printf(参数 1, 参数 2, …, 参数 n);
```

函数中的每一个参数按照给定的格式和顺行依次输出。例如，显示一个字符型变量和整型变量：

```
printf("the Int is %d,the Char is %c",iInt,cChar);
```





表 3.6 中列出了有关 printf 函数的格式字符。

表 3.6　printf 函数的格式字符

格 式 字 符	功 能 说 明
d,i	以带符号的十进制形式输出整数（整数不输出符号）
o	以八进制无符号形式输出整数
x,X	以十六进制无符号形式输出整数。用 x 输出十六进制数的 a～f 时以小写形式输出；用 X 时，则以大写字母输出
u	以无符号十进制形式输出整数
c	以字符形式输出，只输出一个字符
s	输出字符串
f	以小数形式输出
e,E	以指数形式输出实数，用 e 时指数以“e”表示，用 E 时指数以“E”表示
g,G	选用%f 或%e 格式中输出宽度较短的一种格式，不输出无意义的 0。若以指数形式输出，则指数以大写表示

【例 3.5】 使用格式输出函数 printf 输出不同类型的变量。

在本实例中，使用 printf 函数对不同类型变量进行输出，对使用 printf 函数所用到的输出格式进行分析理解。

实例位置：光盘\MR\Instance\03\3.5

```
#include “stdafx.h”
int main()
{
	int iInt=10;                                    //定义整型变量
	char cChar='A';                                 //定义字符型变量
	float fFloat=12.34f;                            //定义单精度浮点型
	printf("the int is: %d\n",iInt);                //使用 printf 函数输出整型
	printf("the char is: %c\n",cChar);              //输出字符型
	printf("the float is: %f\n",fFloat);            //输出浮点型
	printf("the stirng is: %s\n","I LOVE YOU");     //输出字符串
	return 0;
}
```

在程序中定义一个整型变量 iInt，在 printf 函数中使用格式字符%d 进行输出。字符型变量 cChar 赋值为‘A’，在 printf 函数中使用格式字符%c 输出字符。格式字符%f 用来输出实型变量的数值。在最后一个 printf 输出函数中，可以看到使用%s 将一个字符串进行输出，字符串不包括双引号。

运行程序，显示效果如图 3.5 所示。

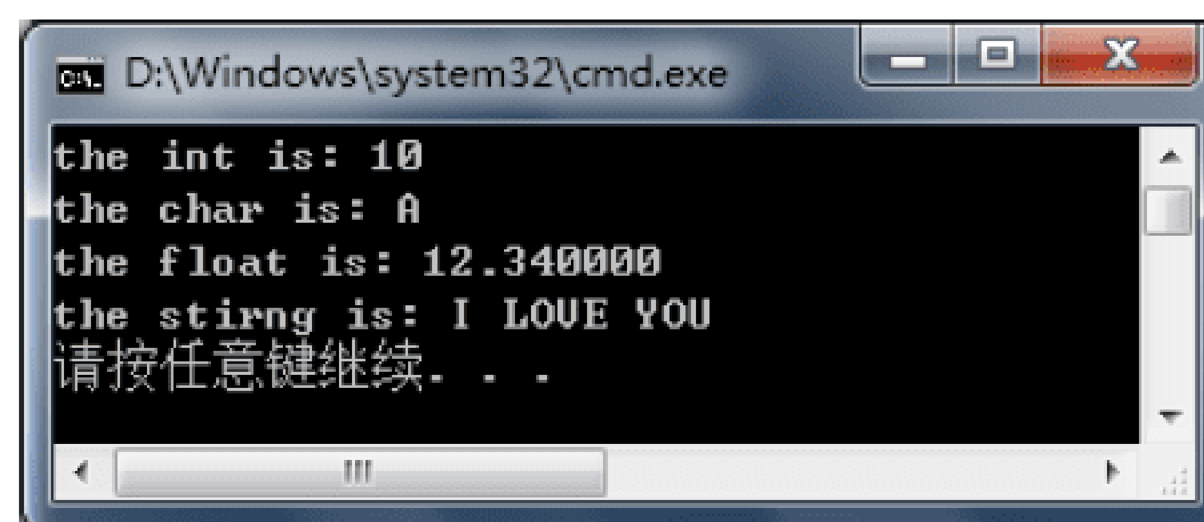


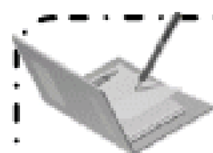
图 3.5　使用格式输出函数 printf

另外，在格式说明中，在%符号和上述格式字符间可以插入几种附加符号，如表 3.7 所示。


Note

表 3.7　printf 的附加格式说明字符

字　　符	功 能 说 明
字母 l	用于长整型整数，可加在格式符 d、o、x、u 前面
m（代表一个整数）	数据最小宽度
n（代表一个整数）	对实数，表示输出 n 位小数；对字符串，表示截取的字符个数
-	输出的数字或字符在域内向左靠

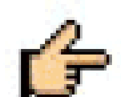

说明：

在使用 printf 函数时，除 X、E、G 外，其他格式字符必须用小写字母，如%d 不能写成%D。

如果想输出“%”符号，则在格式控制处使用%%进行输出即可。

【例 3.6】 在 printf 函数中使用附加符号。

在本实例中，使用 printf 函数的附加格式说明字符，对输出的数据进行更为精准的格式设计。

 **实例位置：光盘\MR\Instance\03\3.6**

```
#include "stdafx.h"
int main()
{
	long iLong=100000;							//定义长整型变量，为其赋值
	printf("the Long is %ld\n",iLong);				//输出长整型变量
	printf("the string is: %s\n","LOVE");			//输出字符串
	printf("the string is: %10s\n","LOVE");			//使用 m 控制输出列
	printf("the string is: %-10s\n","LOVE");			//使用-表示向左靠拢
	printf("the string is: %10.3s\n","LOVE");		//使用 n 表示取字符数
	printf("the string is: %-10.3s\n","LOVE");
	return 0;
}
```

（1）在程序代码中，定义的长整型变量在使用 printf 对其进行输出时，应该在%d 格式字符中添加 l 字符，继而输出长整型变量。

（2）%s 用来输出一个字符串的格式字符，在结果中可以看到输出了字符串“LOVE”。

（3）%10s 为格式%ms，表示输出字符串占 m 列，如果字符串本身长度大于 m，则突破 m 的限制，将字符串全部输出；若字符串小于 m，则用空格进行左补齐。可以看到在字符串“LOVE”前后存在 6 个空格。

（4）%-10s 格式为%-ms，表示如果字符串长度小于 m，则在 m 列范围内，字符串向左靠，右补空格。

（5）%10.3s 格式为%m.ns，表示输出占 m 列，但只取字符串左端 n 个字符。这 n 个字符输出在 m 列的右侧，左补空格。

（6）%-10.3s 格式为%-m.ns，其中 m、n 含义同上，n 个字符输出在 m 列范围内的左侧，右补空格。如果 n>m，则 m 自动取 n 值，即保证 n 个字符正常输出。

运行程序，显示效果如图 3.6 所示。





Note

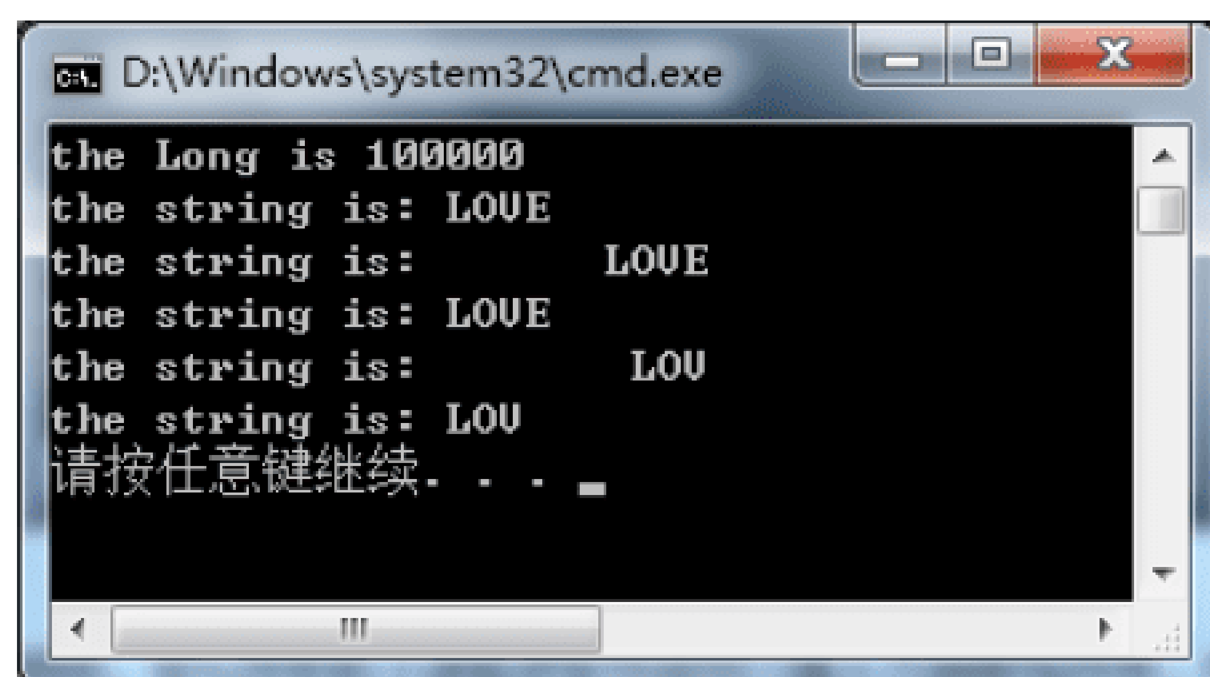


图 3.6　在 printf 函数中使用附加符号

3.4.2　利用 scanf 格式输入数据

与格式输出函数 printf 相对应的是格式输入函数 scanf。该函数的功能是指定固定的格式，并且按照指定的格式接收用户在键盘上输入的数据，最后将数据存储在指定的变量中。

scanf 函数的一般格式为：

```
scanf(格式控制, 地址列表);
```

通过 scanf 函数的一般格式可以看出，参数位置中的格式控制与 printf 函数相同。例如，%d 表示十进制的整型，%c 表示单字符。而在地址列表中，此处应该给出用来接收数据变量的地址。例如得到一个整型数据的操作：

```
scanf("%d",&iInt);                          //得到一个整型数据
```

在上面的代码中，&符号表示取 iInt 变量的地址，因此不用关心变量的地址具体是多少，只要在代码中在变量的标识符前加&符号，就表示取变量的地址。



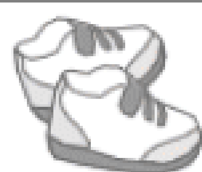
说明：

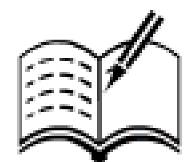
编写程序时，在 scanf 函数参数的地址列表处，一定要使用变量的地址，而不是变量的标识符，否则编译器会提示出现错误。

表 3.8 中列出了 scanf 函数中使用的格式字符。

表 3.8　scanf 函数格式字符

格 式 字 符	功 能 说 明
d,i	用来输入有符号的十进制整数
u	用来输入无符号的十进制整数
o	用来输入无符号的八进制整数
x,X	用来输入无符号的十六进制整数（大小写作用是相同的）
c	用来输入单个字符
s	用来输入字符串
f	用来输入实型，可以用小数形式或指数形式输入
e,E,g,G	与 f 作用相同，e 与 f、g 之间可以相互替换（大小写作用相同）





Note

说明：

格式字符%s 用来输入字符串。将字符串送到一个字符数组中，在输入时以非空白字符开始，以第一个空白字符结束。字符串以串结束标志“\0”作为最后一个字符。

【例 3.7】 使用 scanf 格式输入函数得到用户输入的数据。

在本实例中，利用 scanf 函数得到用户输入的两个整型数据，因为 scanf 函数只能用于输入操作，所以若在屏幕上显示信息则使用显示函数。

实例位置：光盘\MR\Instance\03\3.7

```
#include "stdafx.h"
int main()
{
	int iInt1,iInt2;                                    //定义两个整型变量
	puts("Please enter two numbers:");                  //通过 puts 函数输出提示信息的字符串
	scanf("%d%d",&iInt1,&iInt2);                        //通过 scanf 函数得到输入的数据
	printf("The first is : %d\n",iInt1);                //显示第一个输入的数据
	printf("The second is : %d\n",iInt2);               //显示第二个输入的数据
	return 0;
}
```

（1）为了能接收用户输入的整型数据，在程序代码中定义了两个整型变量 iInt1 和 iInt2。

（2）因为 scanf 函数只能接收用户的数据，而不能显示信息，所以先使用 puts 函数输出一段字符表示信息提示。puts 函数在输出字符串之后会自动进行换行，这样就可以省去使用换行符。

（3）调用 scanf 格式输入函数，在函数参数中可以看到，在格式控制的位置使用双引号将格式字符包括，%d 表示输入的为十进制的整数。在参数中的地址列表位置，使用&符号表示变量的地址。

（4）此时变量 iInt1 和 iInt2 已经得到了用户输入的数据，调用 printf 函数将变量进行输出，这里要注意区分的是，printf 函数使用的是变量的标识符，而不是变量的地址。scanf 函数使用的是变量的地址，而不是变量的标识符。

说明：

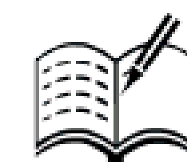
程序是怎样将输入的内容分别保存到指定的两个变量中的呢？原来，scanf 函数使用空白字符分隔输入的数据，这些空白字符包括空格、换行、制表符（tab）。例如，在本程序中，使用换行作为空白字符。

运行程序，显示效果如图 3.7 所示。



图 3.7　使用 scanf 函数得到用户输入的数据





在 printf 函数中除了格式字符外还有附加格式用于更为具体的说明，相应地，scanf 函数中也有附加格式用于更为具体的格式说明，如表 3.9 所示。

表 3.9　scanf 函数的附加格式

字　　符	功 能 说 明
l	用于输入长整型数据（可用于%ld、%lo、%lx、%lu）以及 double 型的数据（%lf 或%le）
h	用于输入短整型数据（可用于%hd、%ho、%hx）
n（整数）	指定输入数据所占宽度
*	表示指定的输入项在读入后不赋给相应的变量

【例 3.8】 使用附加格式说明 scanf 函数的格式输入。

在本实例中，将所有 scanf 附加格式都进行格式输入的说明，通过这些指定格式的输入后，对比输入前后的结果，观察其附加格式的效果。

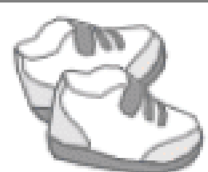
实例位置：光盘\MR\Instance\03\3.8

```
#include "stdafx.h"
int main()
{
	long iLong;                                                     //长整型变量
	short iShort;                                                   //短整型变量
	int iNumber1=1;                                                 //整型变量，为其赋值为 1
	int iNumber2=2;                                                 //整型变量，为其赋值为 2
	char cChar[10];                                                 //定义字符数组变量
	printf("Enter the long integer\n");                             //输出信息提示
	scanf("%ld",&iLong);                                            //输入长整型数据
	printf("Enter the short integer\n");                            //输出信息提示
	scanf("%hd",&iShort);                                           //输入短整型数据
	printf("Enter the number:\n");                                  //输出信息提示
	scanf("%d*%d",&iNumber1,&iNumber2);                             //输入整型数据
	printf("Enter the string but only show three character\n");    //输出信息提示
	scanf("%3s",cChar);                                             //输入字符串
	printf("the long interger is: %ld\n",iLong);                    //显示长整型值
	printf("the short interger is: %hd\n",iShort);                  //显示短整型值
	printf("the Number1 is: %d\n",iNumber1);                        //显示整型 iNumber1 的值
	printf("the Number2 is: %d\n",iNumber2);                        //显示整型 iNumber2 的值
	printf("the three character are: %s\n",cChar);                 //显示字符串
	return 0;
}
```

（1）为了程序中的 scanf 函数能接收数据，在程序代码中定义所使用的变量。为了演示不同格式说明的情况，定义变量的类型有长整型、短整型和字符数组。

（2）使用 printf 函数显示一串字符，提示输入的数据为长整型，调用 scanf 函数使变量 iLong 得到用户输入的数据。在 scanf 函数的格式控制部分，格式字符使用 l 附加格式表示的为长整型。

（3）再使用 printf 函数显示数据提示，提示输入的数据为短整型。调用 scanf 函数时，使用附加格式字符 h 表示短整型。



（4）使用格式字符“*”的作用是表示指定的输入项在读入后不赋给相应的变量，在代码中分析这句话的含义就是，第一个%d 是输入 iNumber1 变量，第二个%d 是输入 iNumber2 变量，但是在第二个%d 前有一个“*”附加格式说明字符，这样第二个输入的值被忽略，也就是说，iNumber2 变量不保存相应输入的值。



Note

（5）%s 是用来表示字符串的格式字符，将一个数 n（整数）放入%s 中间，这样就指定了数据的宽度。在程序中，scanf 函数中指定的数据宽度为 3，那么在输入一个字符串时，只接收前 3 个字符。

（6）最后利用 printf 函数将输入得到的数据进行输出。

运行程序，显示效果如图 3.8 所示。

```
D:\Windows\system32\cmd.exe
Enter the long integer
100000
Enter the short integer
10000
Enter the number:
10
Enter the string but only show three character
Wonderful!
the long interger is: 100000
the short interger is: 10000
the Number1 is: 10
the Number2 is: 2
the three character are: Won
请按任意键继续. . .
```

图 3.8　使用附加格式说明 scanf 函数的格式输入

3.4.3　标准 I/O 流

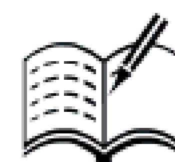
在 C++语言中，数据的输入和输出包括标准输入/输出设备（键盘、显示器）、外部存储介质（磁盘）上的文件和内存的存储空间 3 个方面的输入/输出。对标准输入/输出设备的输入/输出简称为标准 I/O，对在外存磁盘上文件的输入/输出简称文件 I/O，对内存中指定的字符串存储空间的输入/输出简称为串 I/O。

C++语言中把数据之间的传输操作称为流。C++中的流既可以表示数据从内存传送到某个载体或设备中，即输出流；也可以表示数据从某个载体或设备传送到内存缓冲区变量中，即输入流。C++中所有流都是相同的，但文件可以不同（文件流会在后面讲到）。使用流以后，程序用流统一对各种计算机设备和文件进行操作，使程序与设备、文件无关，从而提高了程序设计的通用性和灵活性。

C++语言定义了 I/O 类库供用户使用，标准 I/O 操作有 4 个类对象，它们分别是 cin、cout、cerr 和 clog。其中 cin 代表标准输入设备键盘，也称为 cin 流或标准输入流。cout 代表标准输出显示器，也称为 cout 流或标准输出流，当进行键盘输入操作时使用 cin 流，当进行显示器输出操作时使用 cout 流，当进行错误信息输出操作时使用 cerr 或 clog。

C++的流通过重载运算符“<<”和“>>”执行输入和输出操作。输出操作是向流中插入一个





字符序列，因此，在流操作中，将左移运算符“<<”称为插入运算符。输入操作是从流中提取一个字符序列，因此，将右移运算符“>>”称为提取运算符。

1．cout 语句的一般格式

```
cout<<表达式 1<<表达式 2<<…<<表达式 n;
```

cout 代表着显示器，执行 cout << x 操作就相当于把 x 的值输出到显示器。

先把 x 的值输出到显示器屏幕上，在当前屏幕光标位置显示出来，然后 cout 流恢复到等待输出的状态，以便继续通过插入操作符输出下一个值。当使用插入操作符向一个流输出一个值后，再输出下一个值时将被紧接着放在上一个值的后面，所以为了让流中前后两个值分开，可以在输出一个值后接着输出一个空格，或一个换行符，或是其他所需要的字符或字符串。

一个 cout 语句可以分写成若干行。例如：

```
cout<< "Hello World!" <<endl;
```

可以写成：

```
cout<< "Hello"                          //注意行末尾无分号
<<" "
<<"World!"
<<endl;                                 //语句最后有分号
```

也可写成多个 cout 语句：

```
cout<< "Hello";                         //语句末尾有分号
cout <<" ";                             //输出空格
cout <<"World!.";                       //输出 World
cout<<endl;                             //回车换行
```

以上 3 种情况的输出均正确。

2．cin 语句的一般格式

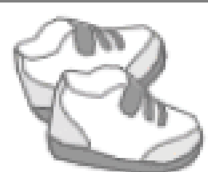
```
cin>>变量 1>>变量 2>>…>>变量 n;
```

cin 代表键盘，执行 cin>>x 就相当于把从键盘输入的数据赋值给变量。

当从键盘上输入数据时，只有当输入完数据并按下回车键后，系统才把该行数据存入到键盘缓冲区，供 cin 流顺序读取给变量。另外，从键盘上输入的每个数据之间必须用空格或回车符分开，因为 cin 为一个变量，读入数据时是以空格或回车符作为其结束标志的。

当 cin>>x 操作中的 x 为字符指针类型时，则要求从键盘的输入中读取一个字符串，并把它赋值给 x 所指向的存储空间，若 x 没有事先指向一个允许写入信息的存储空间，则无法完成输入操作。另外，从键盘上输入的字符串，其两边不能带有双引号定界符，若有则只作为双引号字符看待。对于输入的字符也是如此，不能带有单引号定界符。

cin 函数相当于 c 库函数的 scanf，将用户的输入赋值给变量。示例如下：





Note

```
#include "stdafx.h"
#include <iostream>
using std::cout;
using std:cin;
void main()
{
	int iInput;                                     //定义一个整型变量
	cout << "Please input a number:" <<endl;
	cin >> iInput;                                  //给变量赋值
	cout << "the number is:" << iInput<<endl;
}
```

示例将用户输入的数打印出来。

【例 3.9】 简单输出字符。

实例位置：光盘\MR\Instance\03\3.9

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int i=0;
	cout << i<< endl;                               //输出 i 的值
	cout << "HelloWorld" <<endl;
}
```

程序运行后将向控制台屏幕输出 HelloWorld 字符串，运行效果如图 3.9 所示。

endl 是向流的末尾部位加入换行符。i 是一个整型变量，在输出流中自动将整型变量转换成字符串输出。



图 3.9　向控制台屏幕输出 HelloWorld 字符串

3.4.4　控制输入/输出格式

在头文件<iomanip.h>中定义了一些控制流输出格式的函数，默认情况下整型数按十进制形式输出，也可以通过 hex 将其设置为十六进制输出。流操作的具体控制函数如下。

（1）long setf(long f)

根据参数 f 设置相应的格式标志，返回此前的设置。该参数 f 所对应的实参为无名枚举类型中的枚举常量（又称格式化常量），可以同时使用一个或多个常量，每两个常量之间要用按位或操作符连接。如需要左对齐输出，并使数值中的字母大写时，则调用该函数的实参为 ios::left | ios::uppercase。

（2）long unsetf(long f)

根据参数 f 清除相应的格式化标志，返回此前的设置。如果要清除此前的左对齐输出设置，恢复默认的右对齐输出设置，则调用该函数的实参为 ios::left。

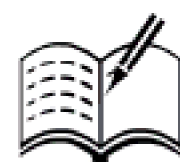


（3）int width()

返回当前的输出域宽。若返回数值为 0，则表明没为刚才输出的数值设置输出域宽。输出域宽是指输出的值在流中所占有的字节数。

（4）int width(int w)

设置下一个数据值的输出域宽为 w，返回为输出上一个数据值所规定的域宽，若无规定则返回 0。注意，此设置不是一直有效，而只是对下一个输出数据有效。



（5）setiosflags(long f)

设置 f 所对应的格式标志，功能与 setf(long f)成员函数相同，当然，在输出该操作符后返回的是一个输出流。如果采用标准输出流 cout 输出它时，则返回 cout。输出每个操作符后都是如此，即返回输出它的流，以便向流中继续插入下一个数据。

（6）resetiosflags(long f)

清除 f 所对应的格式化标志，功能与 unsetf(long f)成员函数相同。输出后返回一个流。

（7）setfill(int c)

设置填充字符的 ASCII 码为 c 的字符。

（8）setprecision(int n)

设置浮点数的输出精度为 n。

（9）setw(int w)

设置下一个数据的输出域宽为 w。

数据输入/输出的格式控制还有更简便的形式，就是使用头文件<iomanip.h>中提供的操作符。使用这些操作符不需要调用成员函数，只要把它们作为插入操作符“”的输出对象即可。

- ☑ dec：转换为按十进制输出整数，是默认的输出格式。
- ☑ oct：转换为按八进制输出整数。
- ☑ hex：转换为按十六进制输出整数。
- ☑ ws：从输出流中读取空白字符。
- ☑ endl：输出换行符\n 并刷新流。刷新流是指把流缓冲区的内容立即写入到对应的物理设备上。
- ☑ ends：输出一个空字符\0。
- ☑ flush：只刷新一个输出流。

【例 3.10】 控制打印格式程序。

实例位置：光盘\MR\Instance\03\3.10

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
	double a=123.456789012345;
	cout << a << endl;
	cout << setprecision(9) << a << endl;
	cout << setprecision(6);				//恢复默认格式（精度为 6）
```

```
    cout << setiosflags(ios::fixed);
    cout << setiosflags(ios::fixed) << setprecision(8) << a << endl;
    cout << setiosflags(ios::scientific) << a << endl;
    cout << setiosflags(ios::scientific) << setprecision(4) << a << endl;
}
```



Note

程序运行结果如图 3.10 所示。

【例 3.11】 整数输出的例子。

实例位置：光盘\MR\Instance\03\3.11

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
    int b=123456;                                           //对 b 赋初值
    cout << b << endl;                                      //输出：123456
    cout << hex << b << endl;                               //输出：1e240
    cout << setiosflags(ios::uppercase) << b << endl;       //输出：1E240
    cout << setw(10) << b <<','<< b << endl;                //输出：1E240，1E240
    cout << setfill('*') << setw(10) << b << endl;          //输出：**** 1E240
    cout << setiosflags(ios::showpos) << b << endl;         //输出：1E240
}
```

程序运行结果如图 3.11 所示。

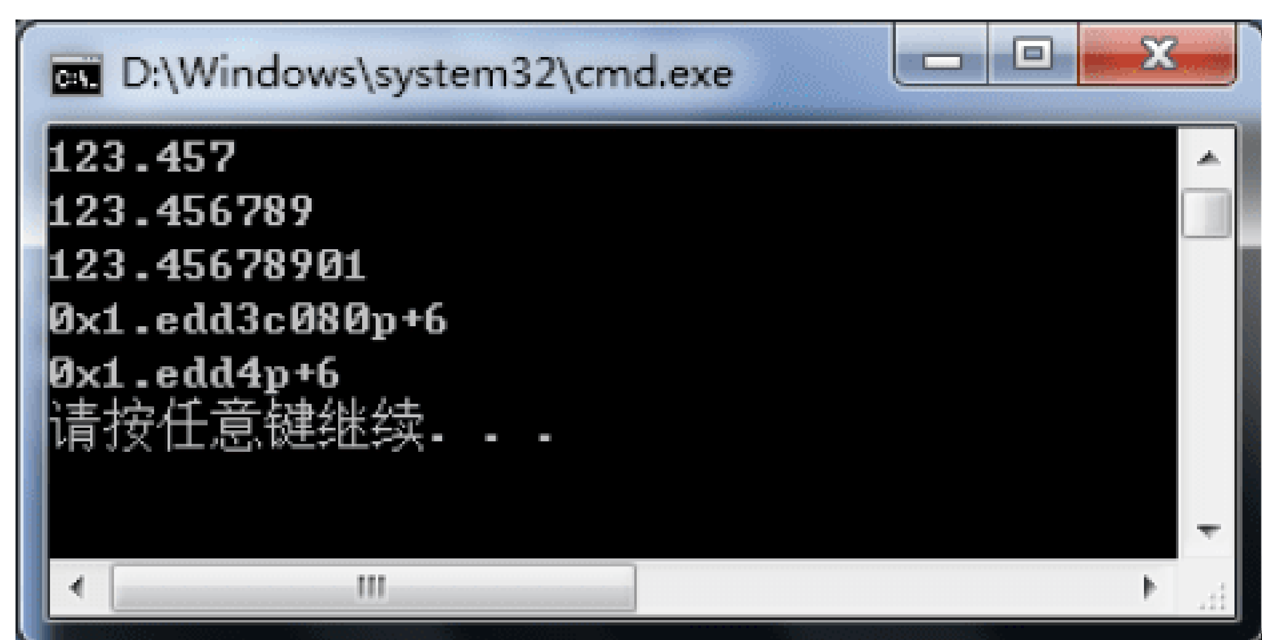


图 3.10　控制打印格式程序



图 3.11　整数输出

【例 3.12】 输出大写的十六进制。

实例位置：光盘\MR\Instance\03\3.12

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
    int i=0x2F,j=255;
    cout << i << endl;                                      //十进制
    cout << hex << i << endl;                               //十六进制 i
```



```
        cout << hex << j << endl;                            //十六进制 j
        cout << hex << setiosflags(ios::uppercase) << j << endl;
}                                                            //控制为十六进制大写格式输出
```

程序执行结果如图 3.12 所示。

【例 3.13】 控制输出精确度。

实例位置：光盘\MR\Instance\03\3.13

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
        int x=123;
        double y=-3.1415;
        cout << "x=";
        cout.width(10);                    //控制输出宽度为 10，不足用空格补全
        cout << x;                         //输出 x 值
        cout << "y=";
        cout.width(10);                    //控制输出宽度为 10，不足用空格补全
        cout << y <<endl;                  //输出 y 值
        cout.setf(ios::left);              //左端对齐
        cout << "x=";
        cout.width(10);                    //控制输出宽度为 10，不足用空格补全
        cout << x;
        cout << "y=";
        cout << y <<endl;
        cout.fill('*');                    //用*填充
        cout.precision(4);                 //四位精度
        cout.setf(ios::showpos);           //显示正负号
        cout << "x=";
        cout.width(10);
        cout << x;
        cout << "y=";
        cout.width(10);                    //控制输出宽度为 10，不足用空格补全
        cout << y <<endl;
}
```

程序运行如图 3.13 所示。



图 3.12　输出大写十六进制

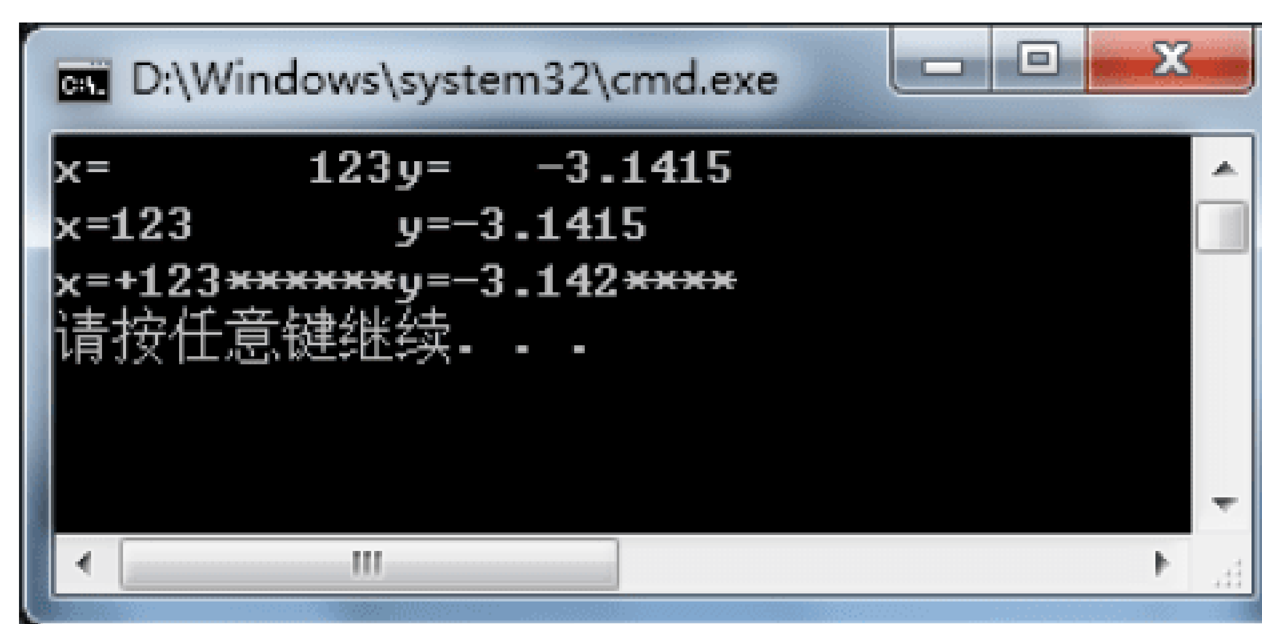


图 3.13　控制输出精确度

Note





Note

【例 3.14】 流输出小数控制。

实例位置：光盘\MR\Instance\03\3.14

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	float x=20,y=-400.00;
	cout << x <<' '<< y << endl;
	cout.setf(ios::showpoint);                    //强制显示小数点和无效 0
	cout << x <<' '<< y << endl;
	cout.unsetf(ios::showpoint);                  //显示小数点
	cout.setf(ios::scientific);                   //设置按科学表示法输出
	cout << x <<' '<< y << endl;
	cout.setf(ios::fixed);                        //设置按定点表示法输出
	cout << x <<' '<< y << endl;
}
```

程序运行结果如图 3.14 所示。

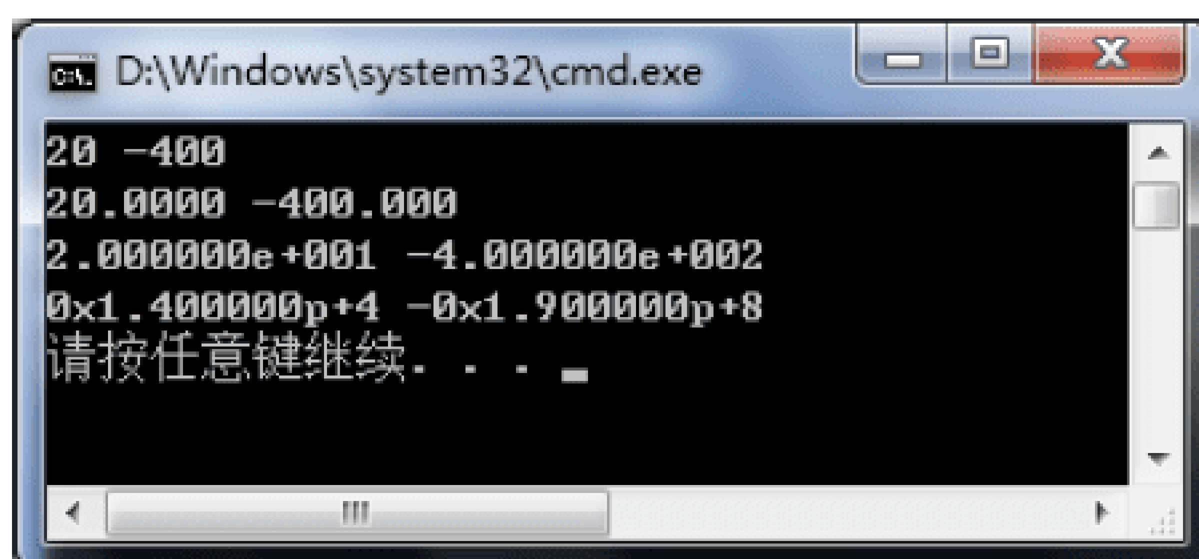


图 3.14　流输出小数控制

3.5　综 合 应 用

3.5.1　计算贷款支付额

本例演示如何编写程序来计算贷款支付额。贷款可以是购车款、学生贷款，或者是房屋抵押贷款。本程序要求用户输入利率、贷款年数和贷款总额，程序计算月支付金额和总偿还金额，并将它们显示出来。

计算月支付额的计算公式如下：

月支付额 ＝(贷款总额*月利率)/(1−1/(1+月利率)年数*12)

【例 3.15】 计算贷款支付额。

实例位置：光盘\MR\Instance\03\3.15

```
#include "stdafx.h"
#include <iostream>
```





```
using namespace std;
int main()
{
	int year;                                                       //贷款年数
	double annualRate;                                              //年利率
	double loanSum;                                                 //贷款总额
	double monthRate;                                               //月利率
	double totalPay;                                                //总支付额
	double monthPay;                                                //月支付额
	cout<<"\n 请输入年贷款利率，如 5.75：";
	cin>>annualRate;                                                //输入年利率
	cout<<"\n 请输入贷款年数，如 15：";
	cin>>year;                                                      //输入贷款年数
	cout<<"\n 请输入贷款总额，如 100000：";
	cin>>loanSum;                                                   //输入贷款总额
	monthRate = annualRate/(12*100);                                //计算月利率
	monthPay = loanSum*monthRate/(1-1/pow(1+monthRate,year*12));    //计算月还款
	totalPay = monthPay*12*year;                                    //计算还款总额
	cout<<"\n 你每月必须偿还："<<monthPay<<"元！";                  //输出月还款
	cout<<"\n 你一共需偿还："<<totalPay<<"元。"<<endl;              //输入还款总额
	return 0;
}
```

程序运行结果如图 3.15 所示。

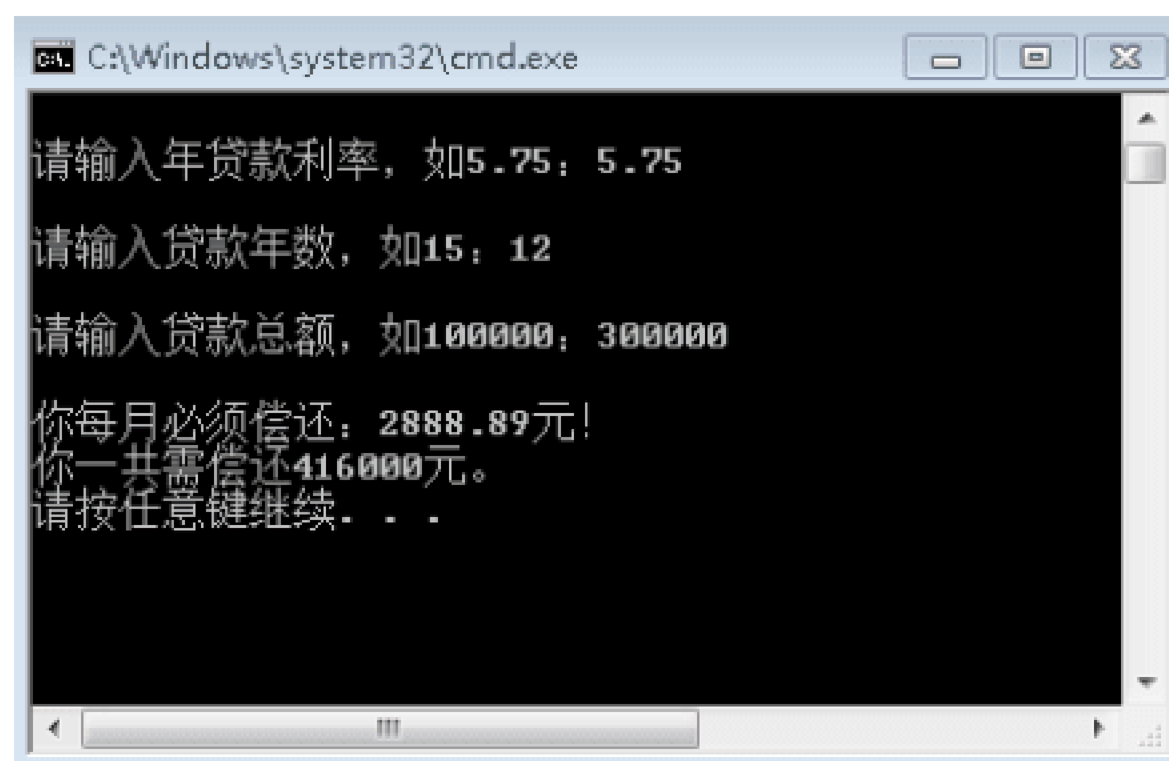


图 3.15　计算贷款支付额

3.5.2　计算函数值

要实现根据用户输入的 x 值，计算函数 y 的值，函数 y 的定义如下：

（1）当 x 大于某一个数 10 时，y = M*x+1。

（2）当 x 小于某一个数 10 时，y =(x+M)*x−3。

【例 3.16】 计算函数值 y。

实例位置：光盘\MR\Instance\03\3.16

```
#include "stdafx.h"
#include <iostream>
```





Note

```
using namespace std;
#define M -1                                    //符号常量中的字母通常采用大写
const int N = 10;                               //定义常量
void main()
{
    int x,y;                                    //定义变量
    cout<<"请输入一个整数：\n";
    cin>>x;                                     //输入 x 的值
    if(x<N)                                     //比较大小，x<N
        y = M*x+1;                              //y 的值
    else                                        //x<N 不成立
        y = (x+M)*x-3;                          //计算 y 的值
    cout<<x<<' '<<y<<endl;                      //输出结果
}
```

程序运行结果如图 3.16 所示。

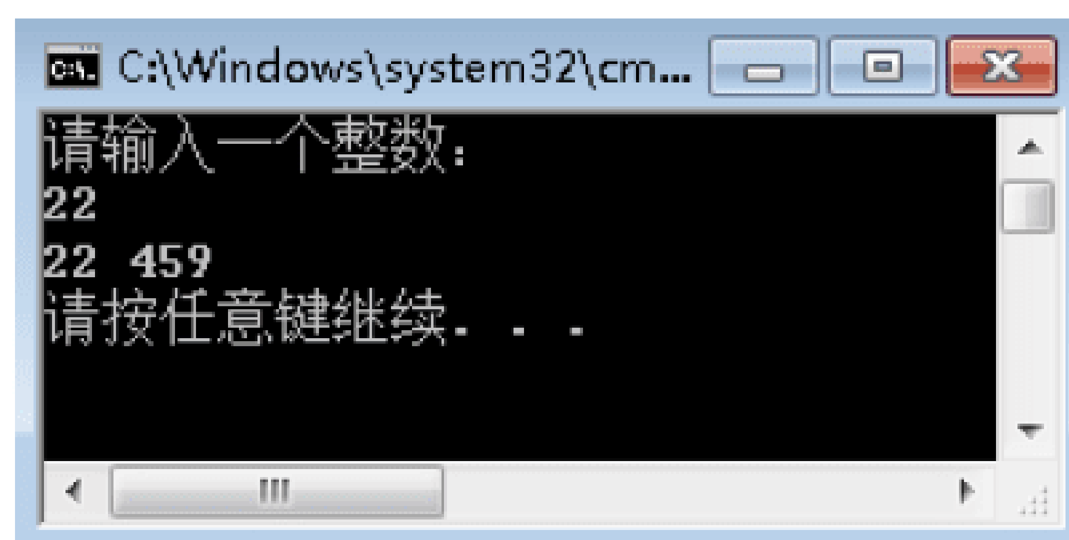


图 3.16 计算函数值 y

3.6 本章常见错误

在程序中最好使用#define 或关键字 const 用符号代表数字常量，符号常量可以加强程序的可读性，使程序更容易维护和修改。

对 const 关键字的总结：

☑ const 修饰只读变量，禁止再次赋值。

☑ 在定义的时候必须初始化。

☑ 存放在静态全局区。在定义的时候编译器不给它分配内存，在第一次调用该变量的时候分配内存，以后不再分配。

3.7 本 章 小 结

本章主要讲述了常量、变量和常用的数据类型，常量、变量和基本数据类型都是计算机语言最基础的部分，读者需仔细理解其基本概念，在以后的程序设计中才能运用自如。本章还介绍了两种流的输出，可以使用流的输出来调试程序，查看输出结果。





3.8　跟我上机

参考答案：光盘\MR\跟我上机

编写一个控制台应用程序，实现如下功能：

（1）提示用户输入半径值。

（2）根据输入的半径值，计算并输出圆的面积。

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main()
{
	float r,S;
	cout<<"请输入半径：";
	cin>>r;                                  //输入半径的值
	if(r<=0)                                 //如果半径不为正数，不能计算
	{
		cout<<"半径不能小于或等于 0\n";
	}
	else                                     //如果半径是正数，用面积公式计算
	{
		S = r*r*3.14;
	}
	cout<<"圆的面积为："<<S<<endl;          //输出圆的面积
	return S;
}
```

第4章

运算符与表达式

（视频讲解：40分钟）

通过鼠标、键盘等设备，将指令发送给计算机，然后计算机将执行结果显示出来，这就是输入与输出。输入设备的信号，显示器上体现的图形、文字，甚至喇叭发出的声音在计算机中的体现都是数据，它是计算机信息的载体。

本章能够完成的主要范例（已掌握的在方框中打勾）

- ☐ 掌握C++数据的基本类型
- ☐ 掌握数据的运算
- ☐ 使用格式输入/输出函数scanf、printf输入/输出信息
- ☐ 使用运算符计算表达式的值
- ☐ 使用sizeof关键字测量数据类型的大小
- ☐ 掌握运算符的结合性和优先级





4.1　C++中的运算符

运算符就是具有运算功能的符号。C++语言中有丰富的运算符，其中有很多运算符都是从 C 语言继承下来的，它新增的运算符有::作用域运算符和->成员指针运算符。

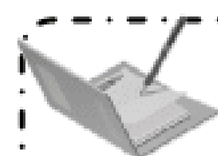
和 C 语言一样，根据使用运算符的对象个数，将运算符分为单目运算符、双目运算符和三目运算符。根据使用运算符的对象之间的关系，将运算符分为算术运算符、关系运算符、逻辑运算符、赋值运算符和逗号运算符等。

4.1.1　算术运算符

算术运算主要指常用的加（+）、减（−）、乘（*）、除（/）四则运算，算术运算符中有单目运算符和双目运算符。算术运算符如表 4.1 所示。

表 4.1　算术运算符

操　作　符	功　　能	目　　数	用　　法
+	加法运算符	双目	expr1 + expr2
−	减法运算符	双目	expr1 – expr2
*	乘法运算符	双目	expr1 * expr2
/	除法运算符	双目	expr1 / expr2
%	模运算	双目	expr1 % expr2
++	自增加	单目	++expr 或 expr++
−−	自减少	单目	−−expr 或 expr−−



说明：

expr 表示使用运算符的对象，可以是表达式、变量和常量。

☑ +是加法运算符，可以进行两个对象的加法运算。

1+1：两个常量相加。

i+1：变量和常量相加。

x+y：两个变量相加。

+100：有符号的常量，强调常量是正数。

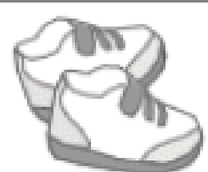
☑ −是减法运算符，可以进行两个对象的减法运算。

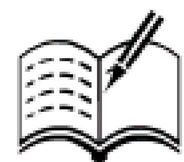
1−1：两个常量相减。

j−1：变量和常量相减。

x−y：两个变量相减。

−100：有符号的常量，强调常量是一个负值。





☑ *是乘法运算符，可以进行两个对象的乘法运算。

2*3：两个常量相乘。

☑ /是除法运算符，可以进行两个对象的除法运算。

2/3：两个常量相除，/运算符左侧的是被除数，也称分子；/运算符右侧的是除数，也称为分母。

在进行除法运算时，除数或分母不可以为 0，为 0 会产生溢出，处理器抛出异常。

2/0：不合法运算。

0/2：合法运算，计算结果是 0。

两个整型数值进行除法运算时返回的结果可能是一个小数，小数点后的数值会被舍去。

☑ %是模运算符，求两个整型的数值或变量在进行除法运算后的余数。

5/2：两个常量进行求模运算，计算结果是 1。

☑ ++是自加运算符，属于单目运算符。有++expr 和 expr++两种形式，++expr 表示 expr 自身加 1 后再进行其他运算；expr++表示 expr 先参加完其他运算后再进行自身加 1，expr 只能是变量。

i++：i 参与运算后，i 的值再自增。

++i：i 自增 1 后再参与其他运算。

1++：不合法。

☑ --是自减运算符，属于单目运算符。有--expr 和 expr--两种形式，--expr 表示 expr 自身减 1 后再进行其他运算；expr--表示 expr 先参加完其他运算后再进行自身减 1，expr 只能是变量。

i--：i 参与运算后，i 的值再自减。

--i：i 自减 1 后再参与其他运算。

1--：不合法。

4.1.2 关系运算符

关系运算主要是对两个对象进行比较，运算结果是逻辑常量真或假。关系运算符如表 4.2 所示。

表 4.2 关系运算符

操 作 符	功 能	目 数	用 法
<	小于	双目	expr1 < expr2
>	大于	双目	expr1 > expr2
>=	大于或等于	双目	expr1 >= expr2
<=	小于或等于	双目	expr1 <= expr2
==	恒等	双目	expr1 == expr2
!=	不等	双目	expr1 != expr2

☑ <是比较两个对象的大小，前者小于后者，运算结果为真。

a<b：两个变量进行比较，如果变量 a 的值小于变量 b 的值，运算结果为真。





Note

2<1：运算结果为假。

☑　>是比较两个对象的大小，前者大于后者，运算结果为真。

a>b：两个变量进行比较，如果变量 a 的值大于变量 b 的值，运算结果为真。

2>1：运算结果为真。

☑　>=是比较两个对象的大小，前者大于或等于后者，运算结果为真。

3>=2：运算结果为真。

2>=2：运算结果为真。

☑　<=是比较两个变量和常量的大小，前者小于或等于后者，运算结果为真。

1<=2：运算结果为真。

☑　==是对两个对象进行判断，前者恒等于后者，运算结果为真。

a==b：两个变量进行比较，如果变量 a 的值恒等于变量 b 的值，运算结果为真。

☑　!=是对两个对象进行判断，前者不等于后者，运算结果为真。

a!=b：两个变量进行比较，如果变量 a 的值不等于变量 b 的值，运算结果为真。

关系运算符都是双目运算符，其结合性均为左结合。关系运算符的优先级低于算术运算符，高于赋值运算符。在 6 个关系运算符中，<、<=、>、>=的优先级相同，高于==和!=，==和!=的优先级相同。

4.1.3　逻辑运算符

逻辑运算符是对真和假这两种逻辑值进行运算，运算后的结果仍是一个逻辑值。逻辑运算符如表 4.3 所示。

表 4.3　逻辑运算符

<table>
<tr><th>操　作　符</th><th>功　　能</th><th>目　　数</th><th>用　　法</th></tr>
<tr><td>&&</td><td>逻辑与</td><td>双目</td><td>expr1 && expr2</td></tr>
<tr><td>||</td><td>逻辑或</td><td>双目</td><td>expr1 || expr2</td></tr>
<tr><td>!</td><td>逻辑非</td><td>单目</td><td>!expr</td></tr>
</table>

☑　&&是对两个对象进行与运算，当两个对象都为真时，结果为真；有一个对象为假或两个对象都为假时，结果为假。

真 && 假：结果为假。

真 && 真：结果为真。

假 && 假：结果为假。

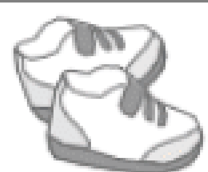
☑　|| 是对两个对象进行或运算，当两个对象都为假时，结果为假，有一个对象为真或两个对象都为真时，结果为真。

真 || 假：结果为真。

真 || 真：结果为真。

假 || 假：结果为假。

☑　!是对一个对象取反运算，当对象为真时，运算结果为假；当对象为假时，运算结果



为真。

!真：运算结果为假。

!假：运算结果为真。

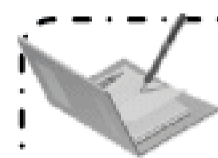
变量 a 和 b 的逻辑运算如表 4.4 所示。



Note

表 4.4　逻辑运算结果

a	b	a&&b	a||b	!a	!b
0	0	0	0	1	1
0	非 0	0	1	1	0
非 0	0	0	1	0	1
非 0	非 0	1	1	0	0



说明：

用 1 代表真，用 0 代表假。

逻辑表达式的对象仍可以是逻辑表达式，从而组成了嵌套的情形。例如，(a||b)&&c 是根据逻辑运算符的左结合性。

【例 4.1】 求逻辑表达式的值。

实例位置：光盘\MR\Instance\04\4.1

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
{
	int i=5,j=8,k=12,l=4,x1,x2;
	x1=i>j&&k>l;                    //表达式 i>j 和 k>1 相与运算
	x2=!(i>j)&&k>l;                 //i>j 取反和 k>1 相与
	printf("%d,%d\n",x1,x2);
}
```

程序运行结果如图 4.1 所示。



图 4.1　运算结果

4.1.4　赋值运算符

赋值运算符分为简单赋值运算符和复合赋值运算符，复合赋值运算符又称为带有运算的赋值运算符，简单赋值运算符就是给变量赋值的运算符。例如：

```
变量=表达式
```

等号（=）就为简单赋值运算符。

C++提供了很多复合赋值运算符，如表 4.5 所示。





表 4.5　复合赋值运算符

操　作　符	功　　能	目　　数	用　　法
+=	加法赋值	双目	expr1 += expr2
−=	减法赋值	双目	expr1 −= expr2
*=	乘法赋值	双目	expr1 *= expr2
/=	除法赋值	双目	expr1 /= expr2
%=	模运算赋值	双目	expr1 % = expr2
<<=	左移赋值	双目	expr1 <<= expr2
>>=	右移赋值	双目	expr1 >>= expr2
&=	按位与运算并赋值	双目	expr1 &= expr2
|=	按位或运算并赋值	双目	expr1 |= expr2
^=	按位异或运算并赋值	双目	expr1 ^= expr2

复合赋值运算符都有等同的简单赋值运算符和其他运算的组合。例如：

a+=b 等价于 a=a+b　　a−=b 等价于 a=a−b　　a*=b 等价于 a=a*b

a/=b 等价于 a=a/b　　a%=b 等价于 a=%b　　a<<=b 等价于 a=a<<b

a>>=b 等价于 a=a>>b　　a&=b 等价于 a=a&b　　a^=b 等价于 a=a^b

a|=b 等价于 a=a|b

复合赋值运算符都是双目运算符，C++采用这种运算符可以更高效地进行加运算，编译器在生成目标代码时能够直接优化，可以使程序代码更小。这种书写形式也非常简洁，使得代码更紧凑。

复合赋值运算符将运算结果返回，作为表达式的值，同时把操作数 1 对应的变量设为运算结果值。例如：

```
int a=6;
a*=5;
```

运算结果是：a 的值为 30。

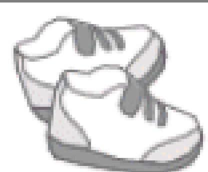
a*=5 等价于 a=a*5，a*5 的运算结果作为临时变量赋给了变量 a。

4.1.5　位运算符

位运算符有位逻辑与、位逻辑或、位逻辑异或和取反运算符，其中位逻辑与、位逻辑或、位逻辑异或为双目运算符，取反运算符为单目运算符。位运算符如表 4.6 所示。

表 4.6　位运算操作符

操　作　符	功　　能	目　　数	用　　法
&	位逻辑与	双目	expr1 & expr2
|	位逻辑或	双目	expr1 | expr2
^	位逻辑异或	单目	expr1 ^ expr2
～	取反运算符	单目	～expr



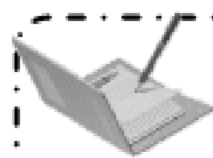


在双目运算符中，位逻辑与优先级最高，位逻辑或次之，位逻辑异或最低。

（1）位逻辑与实际上是将操作数转换成二进制表示方式，然后将两个二进制操作数对象从低位（最右边）到高位对齐，每位求与，若两个操作数对象同一位都为 1，则结果对应位为 1，否则结果中对应位为 0。例如，12 和 8 经过位逻辑与运算后得到的结果是 8。

转为二进制：(0000 0000 0000 1100) & (0000 0000 0000 1000)

```
   0000 0000 0000 1100      （十进制 12 原码表示）
&  0000 0000 0000 1000      （十进制 8 原码表示）
   0000 0000 0000 1000      （十进制 8 原码表示）
```



说明：

十进制在用二进制表示时有原码、反码、补码多种表示方式。

（2）位逻辑或实际上是将操作数转换成二进制表示方式，然后将两个二进制操作数对象从低位（最右边）到高位对齐，每位按位或，若两个操作数对象同一位都为 0，则结果对应位为 0，否则结果中对应位为 1。例如，4 和 8 经过位逻辑或运算后的结果是 12。

转为二进制：(0000 0000 0000 0100) | (0000 0000 0000 1000)

```
   0000 0000 0000 0100      （十进制 4 原码表示）
|  0000 0000 0000 1000      （十进制 8 原码表示）
   0000 0000 0000 1100      （十进制 12 原码表示）
```

（3）位逻辑异或实际上是将操作数转换成二进制表示方式，然后将两个二进制操作数对象从低位（最右边）到高位对齐，每位求异或，若两个操作数对应位相同，则结果为 0，相异则为 1。例如，31 和 22 经过位逻辑异或运算后得到的结果是 9。

转为二进制：(0000 0000 0001 1111) ^ (0000 0000 0001 0110)

```
   0000 0000 0001 1111      （十进制 31 原码表示）
^  0000 0000 0001 0110      （十进制 22 原码表示）
   0000 0000 0001 1111      （十进制 9 原码表示）
```

（4）取反运算符，实际上是将操作数转换成二进制表示方式，然后将各位二进制位由 1 变为 0，由 0 变为 1。例如，41883 取反运算后得到的结果是 23652。

转为二进制：～1010 0011 1001 1011

```
～ 1010 0011 1001 1011      （十进制 41883 原码表示）
   0101 1100 0110 0100      （十进制 23652 原码表示）
```

逻辑位运算符实际上是算术运算符，用该运算符组成的表达式的值是算术值。

4.1.6　移位运算符

移位运算有两个，分别是左移<<和右移>>，这两个运算符都是双目的。

1．左移

左移是将一个二进制操作数对象按指定的移动位数向左移，左边（高位端）溢出的位被丢





弃，右边（低位端）的空位用 0 补充。左移相当于乘以 2 的幂，如图 4.2 所示。

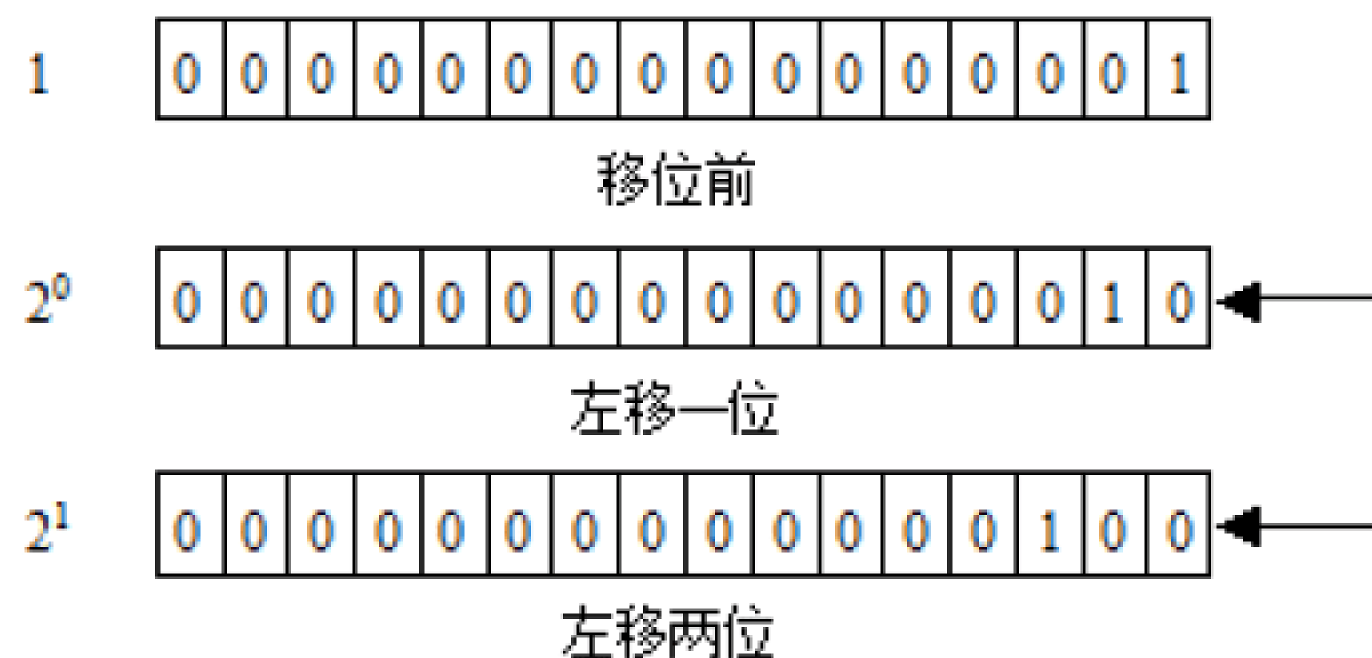


图 4.2　左移位运算

例如，操作数 41883 的二进制是 1010 0011 1001 1011，左移一位变成 18230，左移两位变成 36460，运行过程如图 4.3 所示。

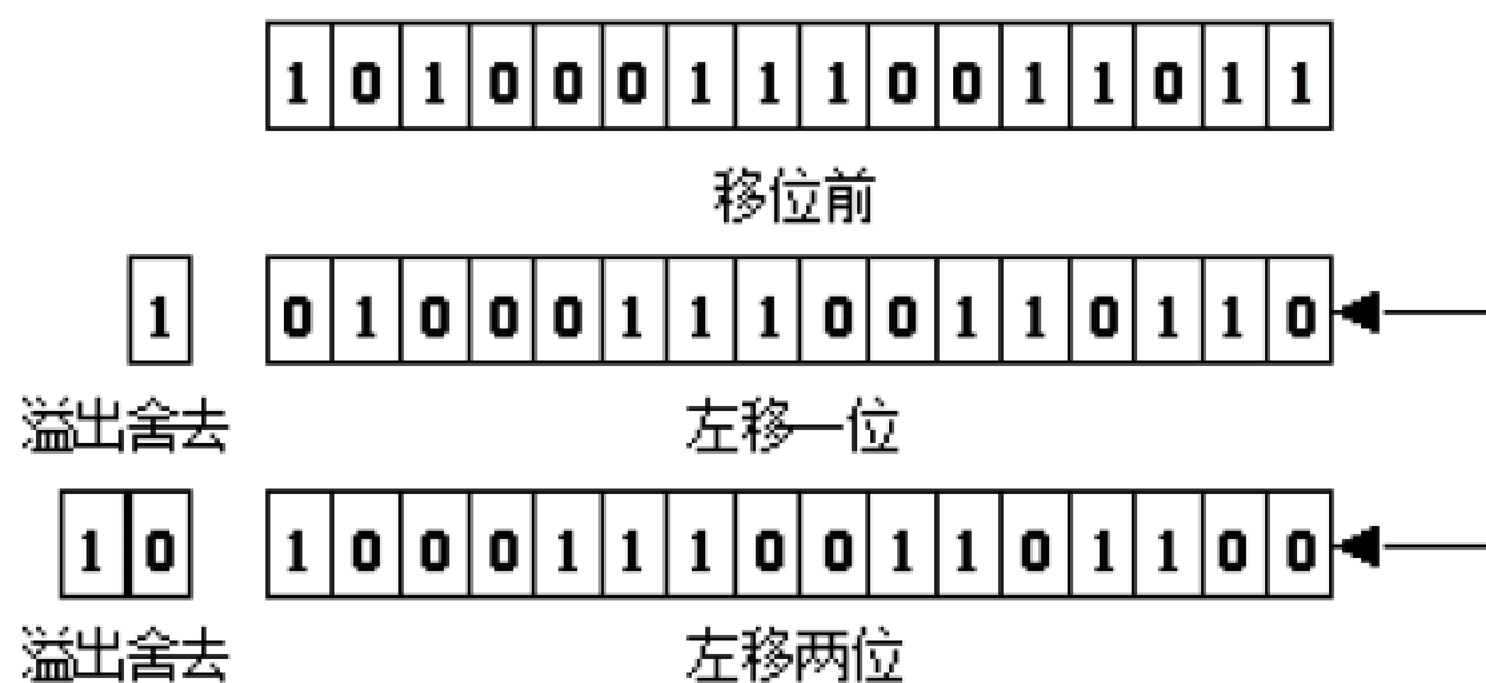


图 4.3　左移位运算过程

【例 4.2】　左移运算。

实例位置：光盘\MR\Instance\04\4.2

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
{
	int a=0x40,b;			//定义整型变量 a 和 b
	b=a<<1;				//a 左移 1 位，结果赋给 b
	cout << b << endl;		//输出 b
}
```

运算结果是：

```
128
```

由于位运算的速度很快，在程序中遇到表达式乘以或除以 2 的幂的情况，一般采用位运算来代替。

2. 右移

右移是将一个二进制的数按指定的位数向右移动，右边（低位端）溢出的位被丢弃，左边（高



位端）的空位或者一律用 0 填充，或者用被移位操作数的符号位填充，运算结果和编译器有关，在使用补码的机器中，正数的符号位为 0，负数的符号位为 1。右移位运算相当于除以 2 的幂，如图 4.4 所示。



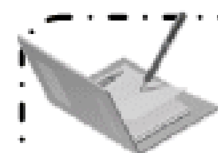


图 4.4　右移位运算

例如，操作数 41883 的二进制是 1010 0011 1001 1011，右移一位变成 20941，右移两位变成 10470，运行过程如图 4.5 所示。



图 4.5　右移位运算过程



说明：

正数右移，低位溢出高位补 0；负数右移分两种，逻辑右移，低位溢出高位补 0；算数右移，低位溢出高位补 1。逻辑右移还是算数右移取决于编译器。

【例 4.3】 右移位运算。

实例位置：光盘\MR\Instance\04\4.3

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	long nWord=0x12345678;
	int nBits;
	nBits=nWord & 0xFFFF;                    //nWord 和 0xFFFF 相与
	printf("low bits are 0x%x\n",nBits);
	nBits=(nWord & 0xFFFF0000)>>16;          //相与再右移 16 位
	printf("hight bits is 0x%x\n",nBits);
}
```



运算结果如图 4.6 所示。

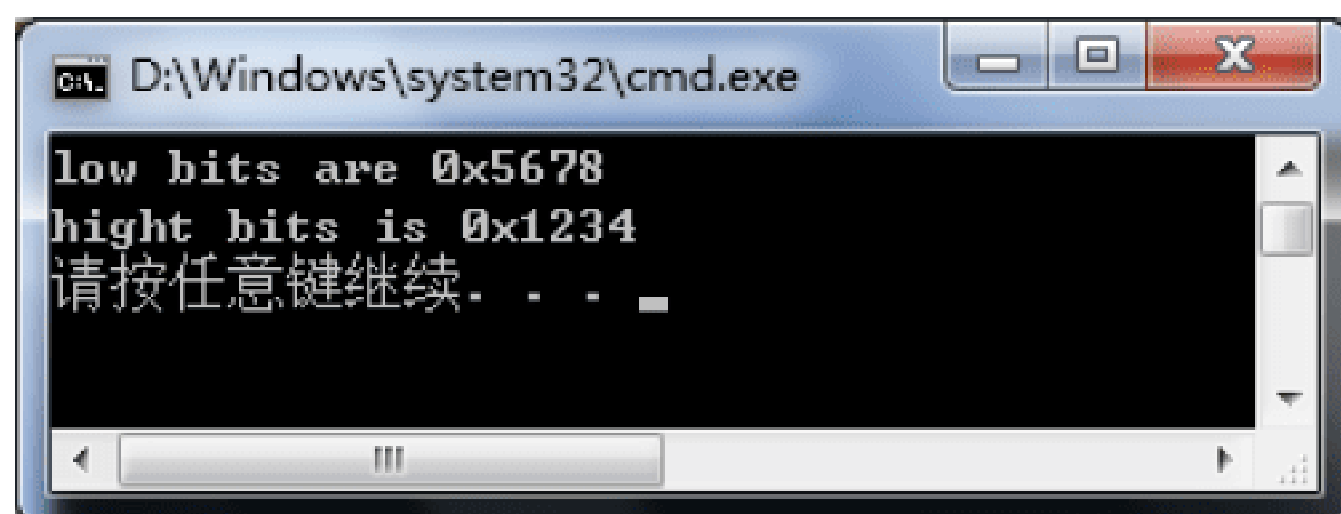


图 4.6　运算结果

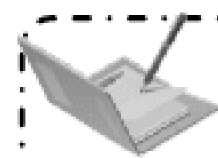


4.1.7　sizeof 运算符

sizeof 是一个很像函数的运算符，也是唯一一个用到字母的运算符。该运算符有两种形式：

```
sizeof(类型说明符)
sizeof(表达式)
```

功能是返回指定的数据类型或表达式值的数据类型在内存中占用的字节数。



说明：

由于 CPU 寄存器的位数不同，同种数据类型占用的内存字节数目就可能不同。

例如：

```
sizeof(char)
```

返回 1，说明 char 类型占用 1 个字节。

```
sizeof(void *)
```

返回 4，说明空指针占用 4 个字节。

```
sizeof(66)
```

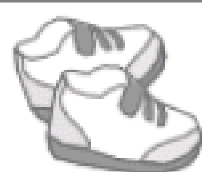
返回 4，说明常量占用 4 个字节。

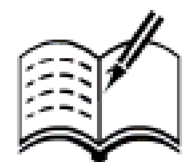
4.1.8　条件运算符

条件运算符是 C++中仅有的一个三目运算符，该运算符需要 3 个运算数对象，形式如下：

```
<表达式 1> ? <表达式 2> : <表达式 3>
```

表示式 1 是一个逻辑值，可以为真或假。若表达式 1 为真，则运算结果是表达式 2，如果表达式 1 为假，则运算结果是表达式 3。这个运算相当于一个 if 语句。





Note

4.1.9　逗号运算符

C 语言中逗号（,）也是一种运算符，称为逗号运算符。逗号运算符的优先级别最低，结合方向自左至右，其功能是把两个表达式连接起来组成一个表达式。逗号运算符是一个多目运算符，并且操作数的个数不限定，可以将任意多个表达式组成一个表达式。例如：

```
x,y,z
a=1,b=2
```

4.2　结合性和优先级

运算符优先级决定了在表达式中各个运算符执行的先后顺序。高优先级运算符要先于低优先级运算符进行运算。例如，根据先乘除后加减的原则，表达式“a+b*c”会先计算 b*c，得到结果再与 a 相加。在优先级相同的情况下，则按从左到右的顺序进行计算。

当表达式中出现了括号时，会改变优先级。先计算括号中的子表达式值，再计算整个表达式的值。

运算符的结合方式有两种：左结合和右结合。左结合表示运算符优先与其左边的标识符结合进行运算，如加法运算；右结合表示运算符优先与其右边的标识符结合，如单目运算符+、-。

同一优先级的运算符优先级别相同，运算次序由结合方向决定。如 1*2/3 中，*和/的优先级别相同，其结合方向自左向右，则等价于(1*2)/3。

运算符的优先级如表 4.7 所示。

表 4.7　运算符优先级

运　算　符	名　　称	优　先　级	结　合　性
() [ ] -> .	圆括号 下标 取类或结构分量 取类或结构成员	1（最高）	→
! ～ ++ -- -	逻辑非 按位取反 自增 1 自减 1 取负	2	←
& * (类型) sizeof	取地址 取内容 强制类型转换 长度计算	2	
* / %	乘 除 整数取模	3	→





Note

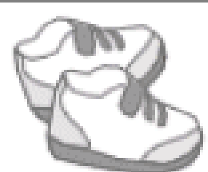
续表

运 算 符	名 称	优 先 级	结 合 性
+ －	加 减	4	→
<< >>	左移 右移	5	→
< <= > >=	小于 小于等于 大于 大于等于	6	→
== !=	恒等 不等于	7	→
&	按位与	8	→
～	按位异或	9	→
|	按位或	10	→
&&	逻辑与	11	→
||	逻辑或	12	→
? :	条件	13	→
=	赋值	14	→
/=	/运算并赋值		←
%=	%运算并赋值		←
*= -= >>= <<= &= ^ |=	*运算并赋值 -运算并赋值 >>运算并赋值 <<运算并赋值 &运算并赋值 ^运算并赋值 | 运算并赋值		
,	逗号（顺序求值）	15（最低）	→

4.3 表 达 式

表达式由运算符、括号、数值对象或变量等几个元素构成。一个数值对象是最简单的表达式，一个表达式可以看作一个数学函数，带有运算符的表达式通过计算将返回一个数值。

```
1 + 1
3.1415926
i + 1
x > y
100 >> 2
j * 3
```



当表达式有两个或多个运算符时，表达式称为复杂表达式，运算符执行的先后顺序由它们的优先级和结合性决定。



Note

```
(X+Y)*Z
a*x+b*y+z
```

一个表达式的值的数据类型由运算符的种类和操作数的数据类型决定。

带运算符的表达式根据运算符的不同，可以分成算术表达式、关系表达式、逻辑表达式、条件表达式和赋值表达式等。

4.3.1　算术表达式

算术表达式的一般形式是：

```
表达式　算术运算符　表达式
```

算术表达式由算术运算符把表达式连接而成，其值的计算很简单，其值的数据类型按下述规定确定：若所有运算符数量类型相同，则表达式运算结果的数据类型和操作数的数据类型相同；若操作数的数据类型不同，就需要转换，表达式运算结果的数据类型取最高的数据类型。

4.3.2　关系表达式

关系表达式的一般形式是：

```
表达式　关系运算符　表达式
```

关系表达式一般只出现在三目运算符、if 语句和循环语句的判断条件中。关系表达式的运算结果都是逻辑型，只能取 true 或 false。数值 0 表示 false，非 0 代表 true。

4.3.3　条件表达式

条件表达式的一般形式是：

```
关系表达式　？　表达式　：表达式
```

条件表达式的值和数据类型取决于?号前表达式的真假，若为真，则整个表达式的运算结果和数据类型和冒号前的操作数相同；若为假，则整个表达式的值与数据类型和冒号后的操作数相同。

4.3.4　赋值表达式

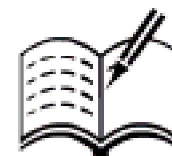
赋值表达式的一般形式是：



```
表达式　赋值运算符　表达式
a>b;
```

赋值运算符的值和数据类型的第一个操作数对象赋值完毕后的值和数据类型相同。

由于赋值运算符的结合性是从右至左，因此可以出现连续赋值的表达式。



Note

4.3.5　逻辑表达式

逻辑表达式的一般形式为：

```
表达式 逻辑运算符 表达式
```

逻辑表达式用逻辑运算符将关系表达式连接起来。逻辑表达式的值也是逻辑型，只能取真值 true 或假值 false。

其中的表达式可以是逻辑表达式，从而组成了嵌套的情形。例如，(a||b)&&c 根据逻辑运算符的左结合性，也可写为 a||b&&c。逻辑表达式的值是式中各种逻辑运算的最后值，以 1 和 0 分别代表“真”和“假”。

逻辑表达式注意事项：

（1）逻辑运算符两侧的操作数，除可以是 0 和非 0 的整数外，也可以是其他任何类型的数据，如实型、字符型等。

（2）在计算逻辑表达式时，只有在必须执行下一个表达式才能求解时，才求解该表达式，也就是说并不是所有的表达式都被求解。

☑　对于逻辑与运算，如果第一个操作数被判定为“假”，系统不再判定或求解第二操作数。

☑　对于逻辑或运算，如果第一个操作数被判定为“真”，系统不再判定或求解第二操作数。

4.3.6　逗号表达式

C 语言中逗号（,）也是一种运算符，称为逗号运算符。逗号运算符的优先级别仅高于赋值运算符，结合方向自左至右，其功能是把两个表达式连接起来组成一个表达式，称为逗号表达式。

其一般形式为：

```
表达式 1，表达式 2
```

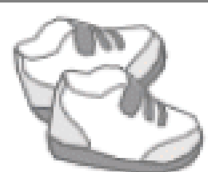
其求值过程是先求解表达式 1，再求解表达式 2，并以表达式 2 的值作为整个逗号表达式的值。

逗号表达式的一般形式可以扩展为：

```
表达式 1，表达式 2，表达式 3，…，表达式 n
```

该逗号表达式的值为表达式 n 的值。

整个逗号表达式的值和类型由最后一个表达式决定。计算一个逗号表达式的值时，从左至右依次计算各个表达式的值，最后计算的一个表达式的值和类型便是整个逗号表达式的值和类型。



逗号表达式的用途仅在于解决只能出现一个表达式的地方却出现多个表达式的问题。

【例 4.4】 逗号运算符的应用。



实例位置：光盘\MR\Instance\04\4.4

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
{
	int a=4,b=6,c=8,res1,res2;
	res1=a,res2=b+c;
	for(int i=0,j=0;i<2;i++)
	{
		printf("y=%d,x=%d\n",res1,res2);
	}
}
```

程序运行结果如图 4.7 所示。

实例中的变量赋初值时、for 循环语句中、printf 打印语句中多处用到了逗号表达式。其中“res1=a,res2=b+c;”比较难理解，res2 等于整个逗号表达式的值，也就是表达式 2 的值，res1 是第一个表达式的值。

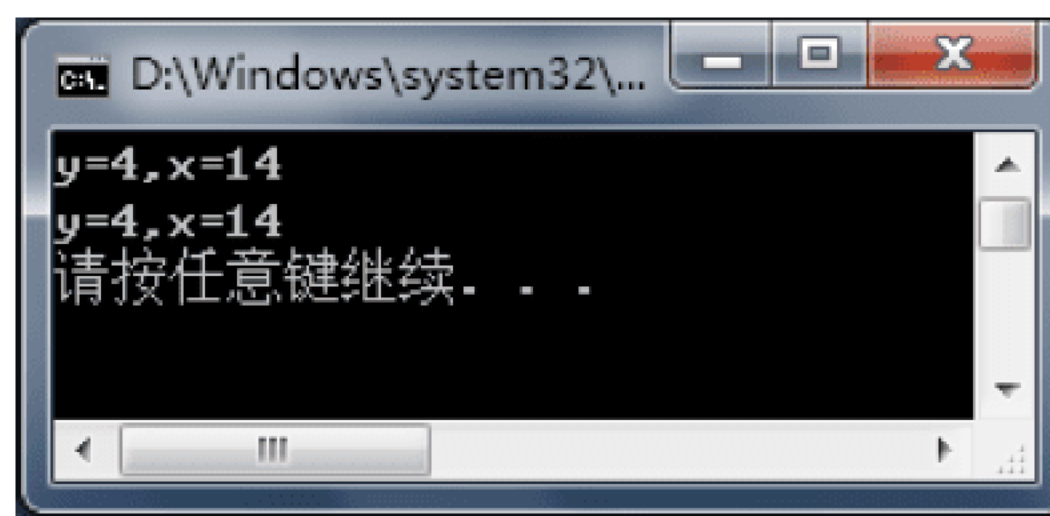


图 4.7　运行结果

逗号表达式的注意事项：

（1）逗号表达式可以嵌套。

```
表达式 1, (表达式 2, 表达式 3)
```

嵌套的逗号表达式可以转换成扩展形式，扩展形式如下：

```
表达式 1, 表达式 2, …, 表达式 n
```

整个逗号表达式的值等于表达式 n 的值。

（2）程序中使用逗号表达式，通常是要分别求逗号表达式内各表达式的值，并不一定要求整个逗号表达式的值。

（3）并不是在所有出现逗号的地方都组成逗号表达式，如在变量说明中，函数参数表中逗号只是用作各变量之间的间隔符。

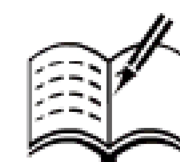
4.3.7　表达式中的类型转换

变量的数据类型转换的方法有两种，一种是隐式转换，一种是强制类型转换。

1．隐式转换

隐式转换发生在不同数据类型的量混合运算时，由编译系统自动完成。隐式转换遵循以下规则：





（1）若参与运算量的类型不同，则先转换成同一类型，然后进行运算。赋值时会把赋值类型和被赋值类型转换成同一类型，一般赋值号右边量的类型将转换为左边量的类型。如果右边量的数据类型长度比左边长时，将丢失一部分数据，这样会降低精度，丢失的部分按四舍五入向前舍入。

（2）转换按数据由低到高顺序执行，以保证精度不降低。

☑ int 型和 long 型运算时，先把 int 量转成 long 型后再进行运算。

☑ 所有的浮点运算都是以双精度进行的，即使仅含 float 单精度量运算的表达式，也要先转换成 double 型，再做运算。

☑ char 型和 short 型参与运算时，必须先转换成 int 型。

类型转换的顺序如图 4.8 所示。

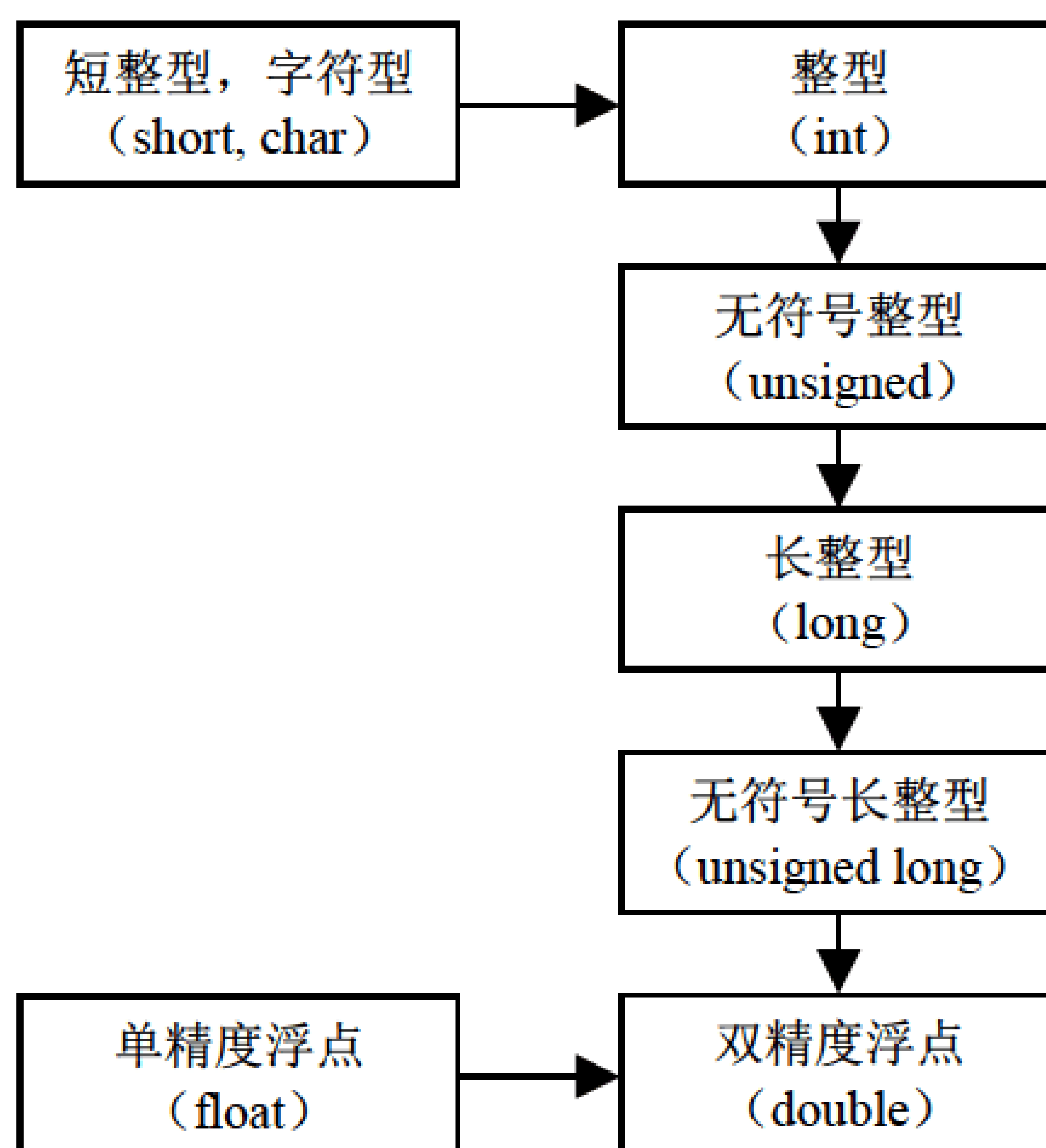


图 4.8　数据类型转换

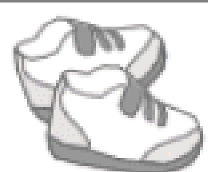
【例 4.5】 隐式类型转换。

实例位置：光盘\MR\Instance\04\4.5

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
{
	double result;
	char a='k';
	int b=10;
	float e=1.515;
	result=(a+b)-e;
	printf("%f\n",result);
}
```

程序运行结果：

```
115.485000
```





Note

2．强制类型转换

强制类型转换是通过类型转换运算来实现的，其一般形式为：

```
类型说明符(表达式)
```

或：

```
(类型说明符)表达式
```

其功能是把表达式的运算结果强制转换成类型说明符所表示的类型。

例如：

```
(float) x;
```

表示把 x 转换为单精度型。

```
(int)(x+y);
```

表示把 x+y 的结果转换为整型。

```
int(1.3)
```

表示一个整数。

强制类型转换后不改变数据说明时对该变量定义的类型。例如：

```
double x;
(int)x;
```

x 仍为双精度类型。

使用强制转换的优点是编译器不必自动进行两次转换，而由程序员负责保证类型转换的正确性。

【例 4.6】 强制类型转换应用。

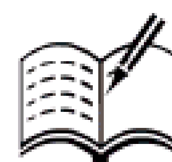
实例位置：光盘\MR\Instance\04\4.6

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
{
	float i,j;
	int k;
	i=60.25;
	j=20.5;
	k=(int)i+(int)j;
	cout << k << endl;
}
```



程序运行结果：

```
80
```



Note

4.4　语 句 概 述

在 C++程序中，语句是最小的可执行单元，一条语句由一个分号结束。

C++程序语句按其功能可以划分为两类，一类是用于描述计算机执行操作运算的，称为操作运算语句；另一类是用于控制操作运算执行顺序的，称为流程控制语句。任何程序设计语句都具备流程控制的功能。基本的控制结构有 3 种：顺序结构、选择结构和循环结构。

顺序结构指按照语句在程序中的先后次序一条一条顺次执行。顺序结构是自然形成的，不需要控制，按默认的顺序执行，顺序控制语句就是一条简单的语句。

1. 表达式语句

表达式语句是由表示式后面加上一个分号组成的。表达式有很多种，如关系表达式、逻辑表达式、算术表达式等，但关系表达式、逻辑表达式多用于循环或选择结构中，只有赋值表达式多用于赋值语句。赋值表达式后面加上一个分号可以形成赋值语句，将右边的表达式（算术表达式）的结果赋给左边的变量。一个赋值语句中可以包含多个赋值表达式。

2. 空语句

空语句只有一个分号，表示什么也不做。空语句经常出现在选择或循环语句中，表示某个分支或循环体不执行具体的操作，也用于编制程序的初始阶段，在搭建程序的模块框架中，先用空语句占位，接下来再逐步细化和补充。

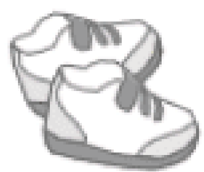
例如：

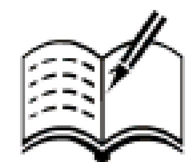
```
while( a < b );
```

上面是一个循环语句，表示当变量 a 小于变量 b 时，在括号中的循环体中要进行什么操作，但不确定循环体应该实现什么功能，所以需要使用空语句占位。空语句语法上是正确的。

3. 复合语句

复合语句是由若干条语句组成的一个集合，它在语法上是一个整体，相当于一个语句，其语法形式是由一对花括号将若干条语句括起来。复合语句经常出现在选择或循环结构中，选择语句的分支和循环语句的循环体由多条语句组成时，用花括号括起来形成一条复合语句，起到层次划分的作用。一个花括号形成了一个范围，这个范围也是变量的作用范围，也可以将花括号内的代码称为程序段。在能使用简单语句的地方，都能够使用复合语句。在一个复合语句中可以包含另外一个或多个复合语句。





例如：

```
{
      x=1;
      y=2;
      a=x+y;
}
```

一个复合语句的花括号外面不能再写分号。

4．函数调用语句

函数由函数名、带实际参数表的圆括号组成，函数调用语句就是在函数后加上一个分号。调用主要指程序执行到函数调用语句时，会跳转到相应的函数体中去执行，执行该函数体中的内容，执行完所有内容后返回到函数调用语句处，执行调用语句下面的语句。可以调用的函数主要有系统库函数和自定义函数。

顺序、选择、循环是结构化程序的 3 种基本结构。选择结构语句、循环结构语句将会在后面章节中讲到。

4.5 判断左值与右值

C++中的每个语句、表达式的结果分为判断左值与右值两类。左值指的是内存当中持续储存的数据，而右值是临时储存的结果。

在程序中，我们声明过的独立的变量，例如：

```
int k;
short p;
char a;
```

它们都是左值。又如：

```
int a = 0;
int b = 2;
int c = 3;
a = c-b;
b = a++;
c = ++a;
c--;
```

c – b 是一个储存表达式结果的临时数据，它的结果将被复制到 a 中，它是一个右值。a++自增的过程实质上是一个临时变量执行了表达式，而 a 的值已经自增了。++a 恰好相反，它是自增之后的 a，是一个左值。由此可见，c--是一个右值。

左值都可以出现在表达式等号的左边，所以成为左值，若表达式的结果不是一个左值。那么表达式的值一定是个右值。





4.6　综 合 应 用

4.6.1　计算三角形周长

【例 4.7】 有 3 根木棍，分别长 3.3cm、4.4cm、5.7cm，将它们搭成一个三角形，设计程序计算并输出这个三角形的周长。用格式化输出语句 printf 输出结果。代码如下：

实例位置：光盘\MR\Instance\04\4.7

```
float a =3.3,b=4.4,c=5.7;                    //定义 3 个浮点型变量并初始化
printf("三角形边长%f\n",a+b+c);               //计算并输出相加的结果
```

运行结果如图 4.9 所示。

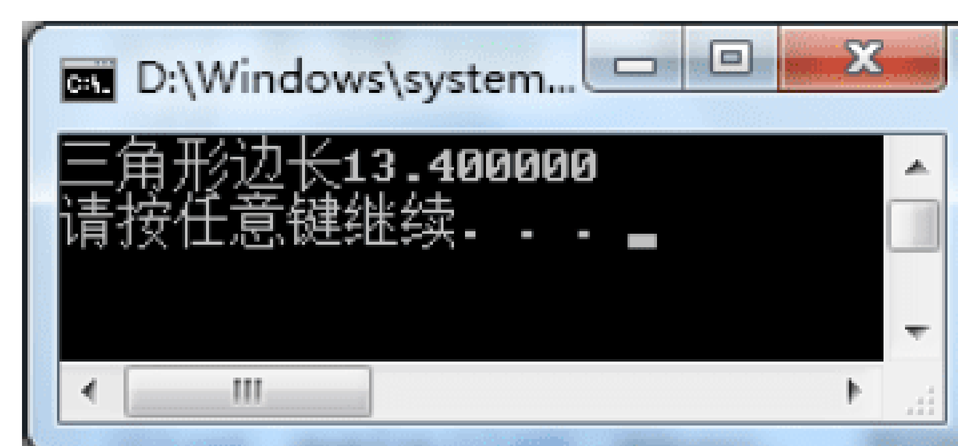


图 4.9　计算三角形周长结果

4.6.2　计算三角形的边长

【例 4.8】 若将例 4.7 中的这些木棍搭建成一个边长分别为 3cm、4cm、5cm 的直角三角形，每个木棍则需要削成相应长度的两段。那么，直角三角形完工后，剩下的 3 根小木棍能否再搭建一个三角形？设计程序判断结果，并输出它（提示：用一个 bool 型变量储存判断结果）。关键代码如下：

实例位置：光盘\MR\Instance\04\4.8

```
#include <iostream>
using namespace std;
int main()
{
    int a1= a*10 - 30, b1= b*10- 40,c1 = c*10-50;          //依据实际精度计算剩下的三条边长
    bool bp1 = (c1+b1>a1)&&(a1+c1>b1)&&(a1+b1>c1);          //判断是否满足两边之和大于第三边
    cout<<bp1<<endl;
    return 0;
}
```

程序结果如图 4.10 所示。

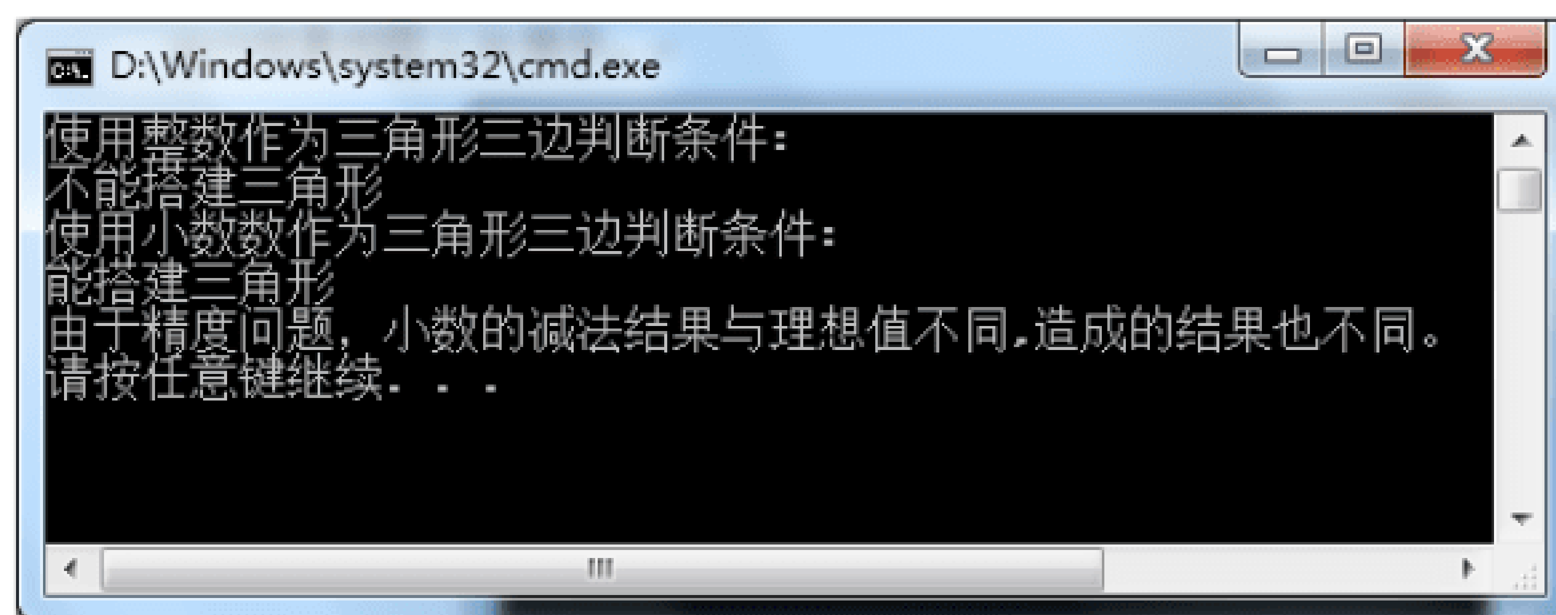
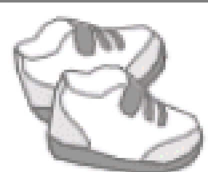


图 4.10　计算三角形边长





Note

4.7 本章常见错误

4.7.1 注意=和==

在比较两个量是否相等时，用关系运算符==，不要粗心误写成赋值号=。如果写成=就意味着把右边的值赋给左边的变量。这种情况可将变量写在右边，如果写成赋值运算符，编译器会报错，提醒程序员。例如：

```
if(0 == a)                                 //如果误写成 0=a，编译出错
```

4.7.2 不要混淆 strlen 和 sizeof

sizeof 是关键字，strlen 是函数，二者都是用来测数据大小的。区别是 strlen 测的是字符串实际占用的空间（不包括结束符），而 sizeof 测的是数据类型占用的固定空间。好比 sizeof 测量的是水杯的容量值，strlen 测的是水杯实际装的水量。例如：

```
char str[50] = {"hello"};                  //定义一个字符型数组，存放字符串 hello
sizeof(str);                               //str 数组的固有大小，50
strlen(str);                               //str 数组里存放实际数据的大小，5 个字节
```

4.7.3 对浮点数求余

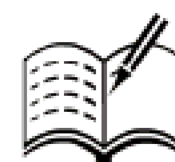
求余运算%要求两个操作数都是整型数。如果对浮点型数做取余操作，会编译出错。例如：

```
int a = 10;
float b = 10.0f;
a%b;                                       //编译出错
```

4.8 本 章 小 结

本章详细介绍了 C++语言中的运算符，以及由运算符组成的表达式和语句，不同运算符有不同的运算规则，掌握这些规则是开发程序的关键。运算符的相关规则关系到程序的运算结果，运算符的优先级是开发人员必须掌握的，学习时要多加注意。与运算符相关的表达式及语句，都是程序的基本组成部分，要理解各语句之间的关系。





4.9　跟 我 上 机

参考答案：光盘\MR\跟我上机

编写一个程序模拟简易计算器功能。任意输入两个整数，求出这两个数的加、减、乘、除等运算结果并输出显示。实现如下：

```
#include <iostream>
using namespace std;
int main()
{
	int num1 = 0;
	int num2 = 0;
	cout<<"输入两个整数:\n";
	cin>>num1>>num2;				//输入要参与计算的两个整数
	char action;					//定义字符变量，存储要执行的计算动作
	cout<<"求和按 a\n 求差按 b\n 求积按 c\n 求商按 d\n";
	cin>>action;
	switch(action)					//判断要执行什么运算操作
	{
	case 'a':
		cout<<num1<<" + "<<num2<<" = "<<num1 + num2<<endl;
		break;
	case 'b':
		cout<<num1<<" - "<<num2<<" = "<<num1 - num2<<endl;
		break;
	case 'c':
		cout<<num1<<" * "<<num2<<" = "<<num1 * num2<<endl;
		break;
	case 'd':
		{
			if(0 == num2)				//判断除数是否为 0
			{
				cout<<"除数不能为 0！ \n";
				return 0;
			}
			else
				cout<< num1<<" / "<< num2<<" = "<< num1 / num2<<endl;
			break;
		}
		default:
			break;
	}
	return 0;
}
```

第 5 章

条件判断语句

（视频讲解：35 分钟）

判断语句是重要的程序控制语句，开发程序时会大量运用。判断语句有很多形式，灵活运用各种形式的判断语句可以提高软件的效率，并且逻辑性强的判断语句容易阅读，可以起到简化代码的作用。

本章能够完成的主要范例（已掌握的在方框中打勾）

- [ ] 掌握 3 种形式的判断语句
- [ ] 了解条件运算符与判断语句的转换
- [ ] 掌握 switch 分支语句
- [ ] 掌握判断语句的嵌套





5.1　决 策 分 支

计算机的主要功能是提供用户计算功能，但在计算的过程中会遇到各种各样的情况，针对不同的情况会有不同的处理方法，这就要求程序开发语言要有处理决策的能力。汇编语言使用判断指令和跳转指令实现决策，高级语言使用选择判断语句实现决策。为描述决策系统的流通，设计人员开发了流程图。流程图使用图形方式描述系统不同状态的不同处理方法。开发人员使用流程图表现程序的结构。

主要的流程图符号如图 5.1 所示。

使用流程图描述十字路口转向的决策，利用方位作决定，判断是否是南方，如果是南方向前行，如果不是南方，寻找南方，如图 5.2 所示。

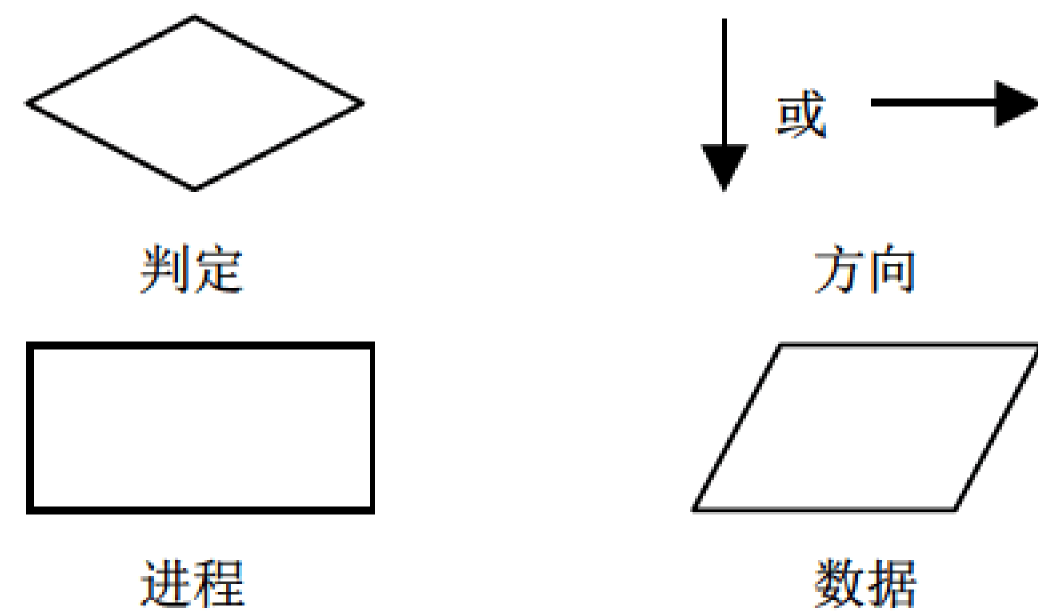


图 5.1　主要的流程图符号

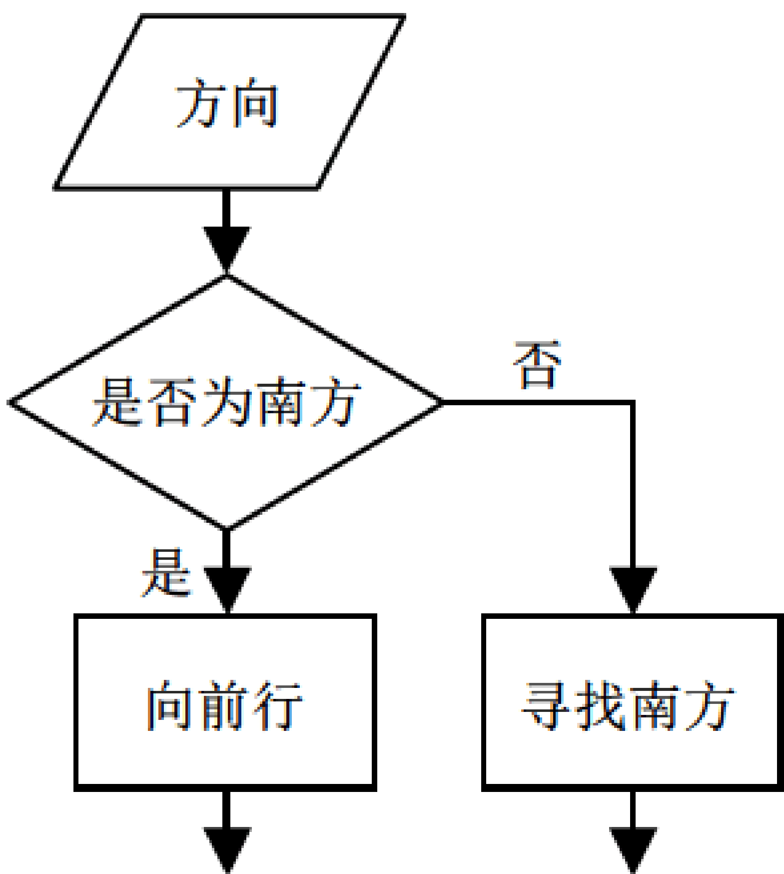


图 5.2　流程图

程序中使用选择判断语句来作决策，选择判断是编程语言的基础语句，在 C++语言中有 3 种选择判断语句，同时提供了 switch 语句，简化多分支决策的处理。下面对选择判断语句进行介绍。

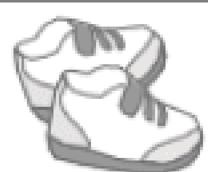
5.2　判 断 语 句

5.2.1　第一种形式的判断语句——if 语句

if 关键字是实现 C++组成判断语句的常用方法，if 语句的形式如下：

```
if(表达式)
语句
```

表达式一般为关系表达式，表达式的运算结果应该是真或假（true 或 false）。如果表达式为真，执行语句，如果表达式的值为假就跳过，执行下一条语句，过程如图 5.3 所示。





Note

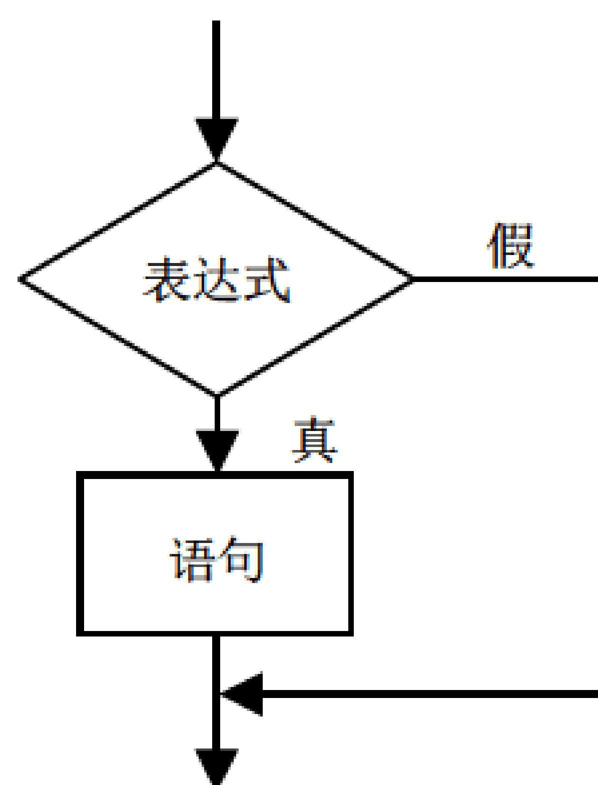


图 5.3 执行 if 语句的流程

【例 5.1】 判断输入数是否为奇数。

实例位置：光盘\MR\Instance\05\5.1

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int iInput;
	cout << "Input a value:" << endl;
	cin >> iInput;				//输入一整型数
	if(iInput%2!=0)				//不能被 2 整除
		cout << "The value is odd number" << endl;
}
```

程序的执行过程如图 5.4 所示。

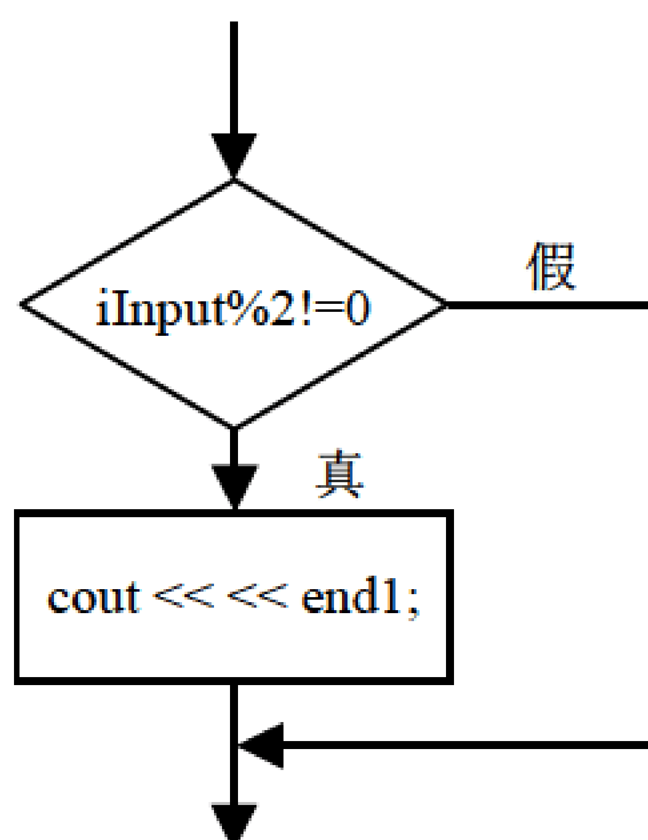


图 5.4 判断奇数的执行过程

程序分两步执行：

（1）定义一个整型变量 iInput，然后使用 cin 获得用户输入的整型数据。

（2）对变量 iInput 的值与 2 进行%运算，如果运算结果不为 0，表示用户输入的是奇数，是奇数就输出字符串“这个整数是奇数”。如果运算结果为 0，则不进行任何输出，程序执行完毕。



说明：

整数与 2 进行%运算，结果只有 0 或 1 两种情况。





要注意第一种形式的判断语句的书写格式。

判断语句：

```
if(a>b)
	max=a;
```

可以写成：

```
if( a>b ) max=a;
```

但不建议使用“if(a>b) max=a;”这种书写方式，这种方式不便于阅读。

判断形式中的语句可以是复合语句，也就是说可以用花括号括起多条简单语句。例如：

```
if(a>b)
{
	tmp=a;
	b=a;
	a=tmp;
}
```

5.2.2　第二种形式的判断语句——if-else 语句

在 if 关键字后，使用 else 关键字表示的是当程序进入到 if-else 语句当中，会根据 if 语句的判断内容，若为真（true），执行 if 语句中的内容；若为假（false），则执行 else 语句的内容，过程如图 5.5 所示。

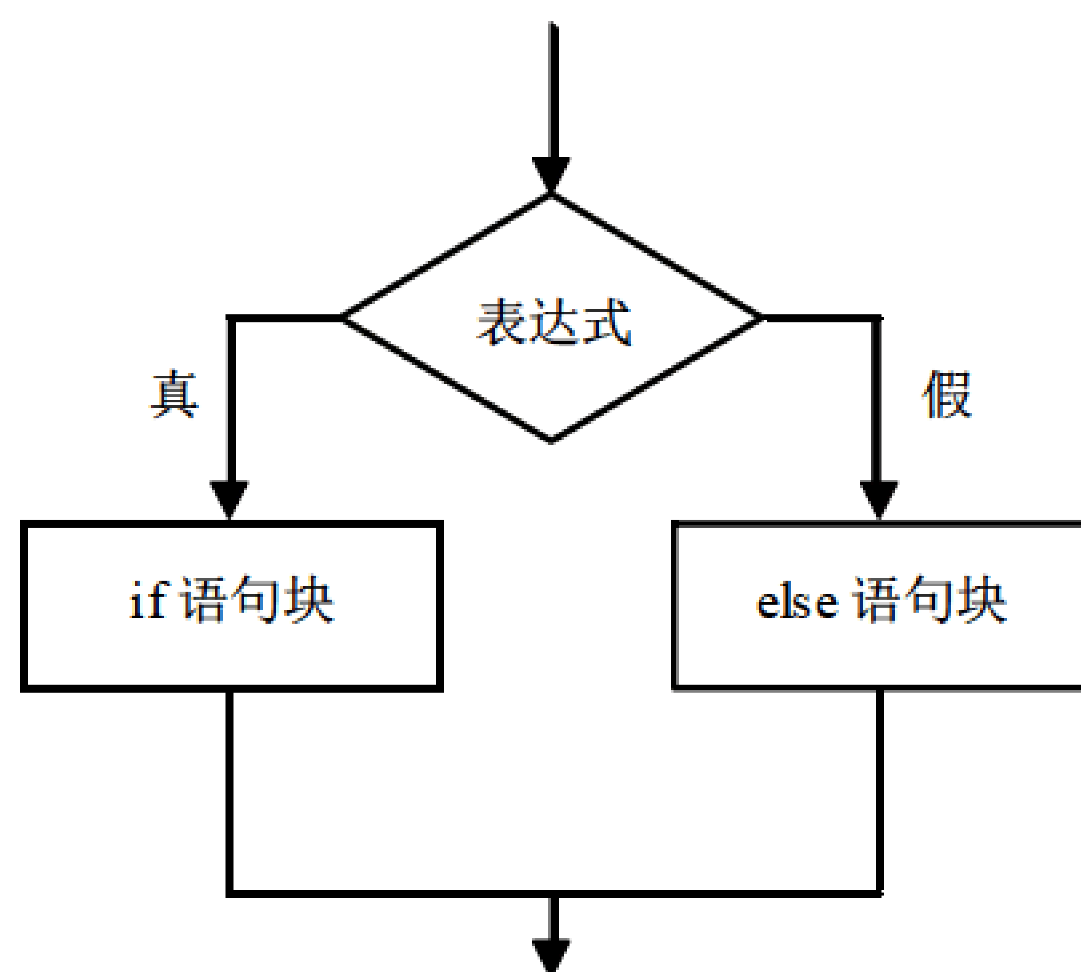
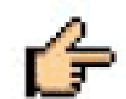
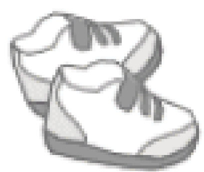


图 5.5　if-else 判断语句过程

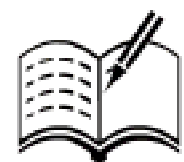
【例 5.2】 根据分数判断是否优秀。

 实例位置：光盘\MR\Instance\05\5.2

```
#include "stdafx.h"
#include <iostream>
using namespace std;
```



```
void main()
{
	int iInput;
	cout<<"大于 90 为优秀成绩"<<endl;
	cout<<"请输入学生成绩"<<endl;
	cin >> iInput;
	if(iInput>90)
		cout << "成绩优秀" << endl;
	else
		cout << "成绩非优秀" << endl;
}
```



Note

用流程图来描述判断语句的执行过程，如图 5.6 所示。

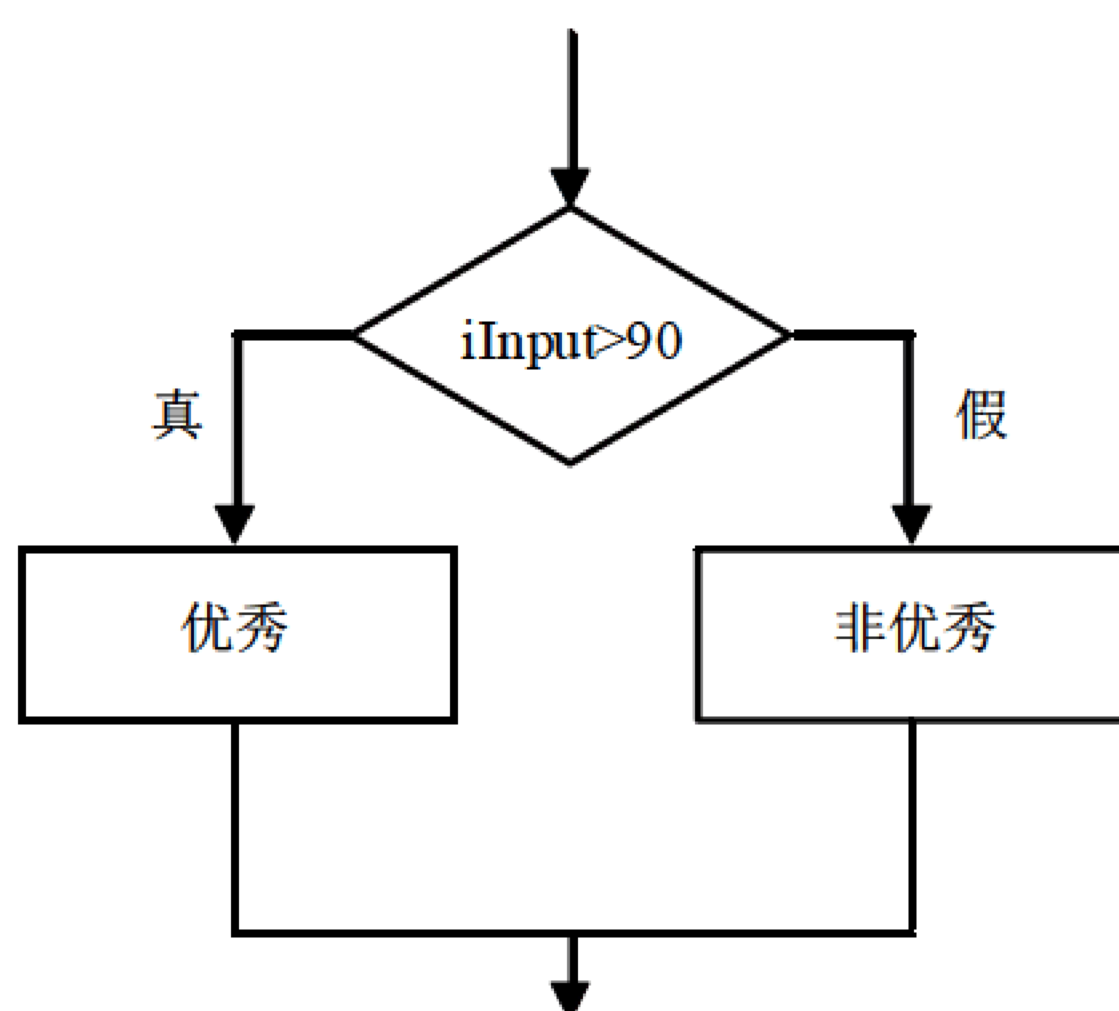


图 5.6　判断语句的执行过程

程序需要和用户交互，用户输入一个数值，将该数值赋值给 iInput 变量，然后判断用户输入的数据是否大于 90，如果大于 90，输出字符串“成绩优秀”，否则输出字符串“成绩非优秀”。

可以看到，程序到此必然经过 if 或者 else 当中的一项。当 else 语句内容为空时，if-else 与 if 语句实现的功能是一样的。

【例 5.3】 if-else 语句的奇偶性判断。

实例位置：光盘\MR\Instance\05\5.3

```
#include <iostream>
using namespace std;
void main()
{
	int iInput;
	cout << "Input a value:" << endl;
	cin >> iInput;	//输入一整型数
	if(iInput%2!=0)
		cout << "这个整数是奇数" << endl;
	else
		cout << "这个整数是偶数" << endl;
}
```





程序分两步执行：

（1）定义一个整型变量 iInput，然后使用 cin 获得用户输入的整型数据。

（2）对变量 iInput 的值与 2 进行%运算，如果运算结果不为 0，表示用户输入的是奇数，是奇数就输出字符串“这个整数是奇数”；如果运算结果为 0，是偶数就输出字符串“这个整数是偶数”，最后程序执行完毕。

使用 else 时的注意事项如下：

☑　else 不能单独使用，必须和关键字 if 一起出现。

else (a>b) max=a 是不合法的。

☑　else 后跟的语句可以是复合语句。

例如：

```
f(a>b)
{
    max=a;
    cout << a << endl;
}
else
{
    max=b;
    cout << b << endl;
}
```

所以实例 5.3 也可视作实例 5.1 的改进版。

5.2.3　第三种形式的判断语句——多次判断语句

在 if 语句中继续使用 if-else 语句，每判断一次就缩小一定的检查范围，它的形式如下：

```
if(表达式 1)
    语句 1;
else if(表达式 2)
    语句 2;
else if(表达式 3)
    语句 3
    …
else if(表达式 m)
    语句 m;
else
    语句 n;
```

表达式一般为关系表达式，表达式的运算结果应该是真或假（true 或 false）。如果表达式为真，执行语句，如果表达式的值为假就跳过，执行下一条语句，执行过程如图 5.7 所示。





Note

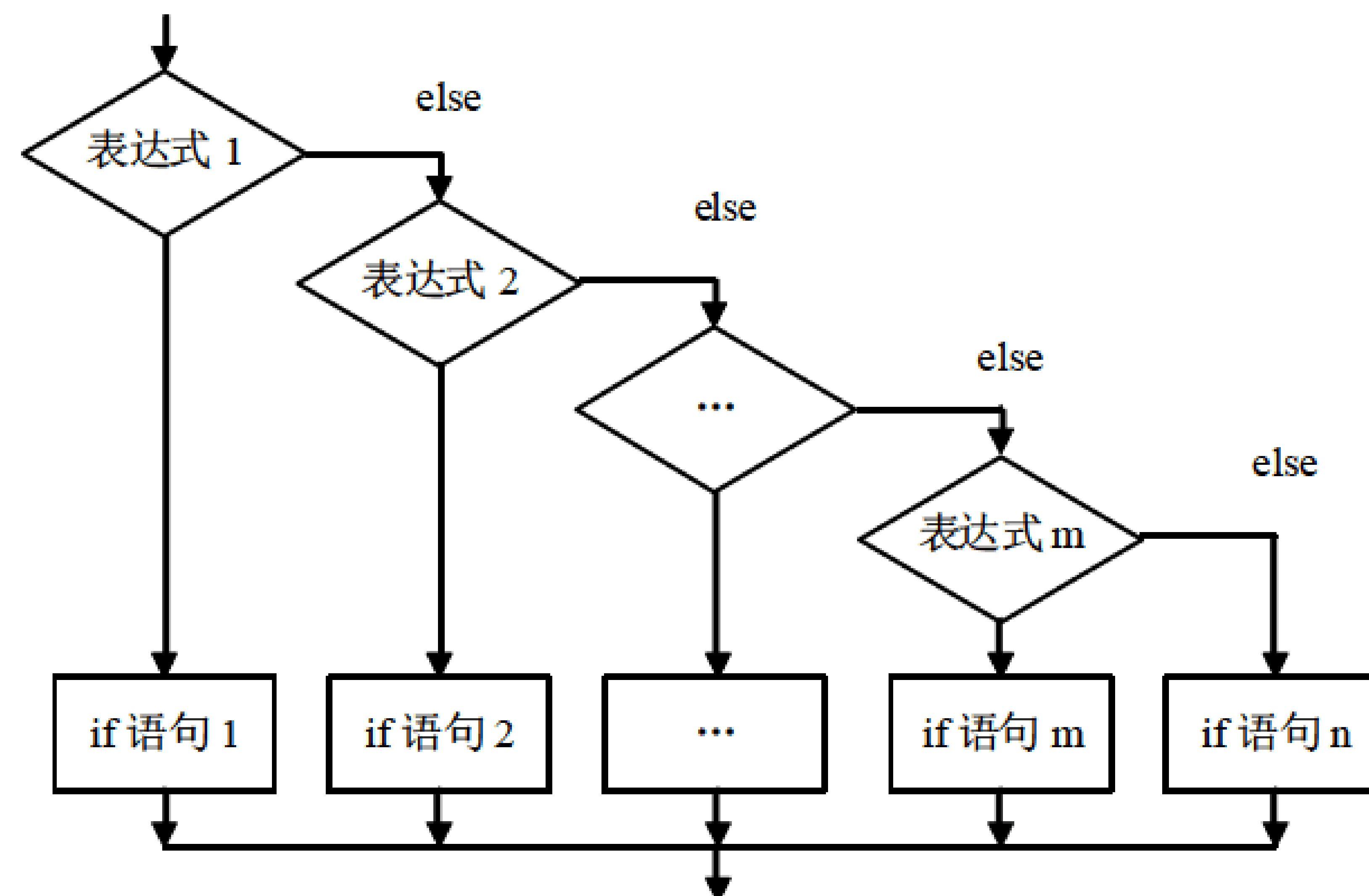


图 5.7　else if 判断语句

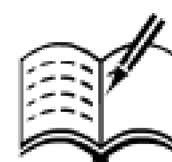
【例 5.4】 根据成绩划分等级。

实例位置：光盘\MR\Instance\05\5.4

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main()
{
	cout<<"输入成绩"<<endl;
	int iInput;
	cin >> iInput;				//输入数据
	if(iInput>=90)				//条件判断
	{
		cout << "优秀" <<endl;
	}
	else if(iInput>=80&& iInput<90)		//else 判断
	{
		cout << "良好" <<endl;
	}
	else if(iInput>=70 && iInput <80)
	{
		cout << "一般" <<endl;
	}
	else if(iInput>=60 && iInput <70)
	{
		cout << "及格" <<endl;
	}
	else if(iInput<60&&iInput>=0)
	{
		cout << "考试不及格，请再加把劲" <<endl;
	}
	return 0;
}
```



程序需要用户输入整型数值，然后判断数值是否大于 90，如果大于 90，输出“优秀”字符串，否则继续判断；判断是否小于 90 大于 80，如果小于 90 大于 80，输出“良好”字符串，否则继续判断；依此类推，最后判断是否小于 60，如果小于 60，输出“考试不及格，请再加把劲”的字符串，最后没有使用 else 再进行判断。



Note

5.3　使用条件运算符进行判断

条件运算符是一个三目运算符，它能像判断语句一样完成判断。例如：

```
max=(iA > iB) ? iA : iB;
```

首先比较 iA 和 iB 的大小，如果 iA 大于 iB 就取 iA 的值，否则取 iB 的值。

可以将条件运算符改为判断语句。例如：

```
if(iA > iB)
	max= iA;
else
	max= iB;
```

【例 5.5】 用条件运算符完成判断数的奇偶性。

实例位置：光盘\MR\Instance\05\5.5

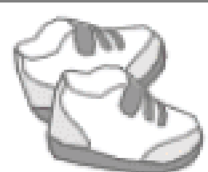
```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
{
	int iInput;
	cout << "输入一个整数" << endl;
	cin >> iInput;						//从键盘中输入一个数
	(iInput%2!=0) ? cout << "这个整数是奇数" : cout << "这个整数是偶数" ;
	cout << endl;
}
```

该程序使用条件运算符完成判断数的奇偶性，比使用判断语句时的代码要简洁。程序同样完成由用户输入整型数，然后和 2 进行%运算，如果运算结果不为 0，是奇数，否则是偶数。

【例 5.6】 判断一个数是否是 3 和 5 的整倍数。

实例位置：光盘\MR\Instance\05\5.6

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
{
	int iInput;
```



```
        cout << "输入一个整数" << endl;
        cin >> iInput;                          //从键盘中输入一个数
        (iInput%3==0 && iInput%5==0)?cout << "yes" : cout<<"no";
        cout << endl;                           //如果能被 3 和 5 整除，输出 yes，否则输出 no
}
```



Note

程序需要用户输入一个整型数，然后用%运算判断能否被 3 和 5 整除，如果同时能被 3 和 5 整除，说明输入的整型数是 3 和 5 的整倍数。

条件运算符可以嵌套，例如：

```
表达式 1?(表达式 a?表达式 b:表达式 c;):表达式 d;
```

【例 5.7】 利用条件表达式判断一个数是否是 3 和 5 的整倍数。

实例位置：光盘\MR\Instance\05\5.7

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
{
        int iInput;
        cout << "输入一个整数" << endl;
        cin >> iInput;                          //从键盘中输入一个数
        (iInput%3==0)?
                ((iInput%5==0) ? cout << "yes" : cout << "no" )
                : cout << "no";
        cout << endl;
}
```

实例 5.6 和实例 5.7 完成同一个目标，都是通过%运算来判断输入的整型数是否是 3 和 5 的整倍数。但实例 5.7 中使用了条件运算符的嵌套，由于条件运算符的嵌套后的代码不容易阅读，一般不建议使用。

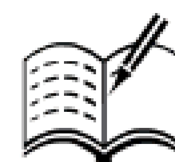
5.4 switch 判断语句

C++语言提供了一种用于多分支选择的 switch 语句。可以使用 if 判断语句做多分支结构程序，但当分支足够多时，if 判断语句会造成代码容易混乱，可读性也很差，如果使用不当就会产生表达式上的错误，所以建议在仅有两个分支或分支数少时使用 if 判断语句，而在分支比较多时使用 switch 语句。

switch 语句的一般形式为：

```
switch(表达式)
{
case  常量表达式 1:
```





```
        语句 1;
        break;
case 常量表达式 2:
        语句 2;
        break;
        …
case 常量表达式 n:
        语句 n;
        break;
default:
        语句 n+1;
}
```

表达式是一个算术表达式，需要计算出表达式的值，该值应该是一个整型数或是一个字符，如果是浮点数，可能会因为精度的不精确而产生错误。

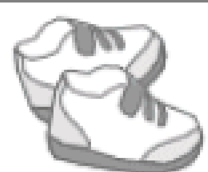
switch 是分支的入口，开始判断是在 case 分语句中，用表达式的值逐一和 case 语句中的值进行比较，有匹配成功的就使用 break；跳出 switch 语句，如果没有匹配成功的，就执行 default 分句。

default 分句是可以不写，如果不写 default 分句，case 分语句中没有匹配成功的就不进行任何操作。

【例 5.8】 根据输入的字符输出字符串。

实例位置：光盘\MR\Instance\05\5.8

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
        cout<<"输入一个 A-D 范围内的大写字母作为成绩评价"<<endl;
        char iInput;
        cin >> iInput;
        switch (iInput)                                   //使用 switch 根据 iInput 选择输出内容
        {
                case 'A':
                        cout << "very good" << endl;
                        break;
                case 'B':
                        cout << "good" << endl;
                        break;
                case 'C':
                        cout << "normal" << endl;
                        break;
                case 'D':
                        cout << "failure" << endl;
                        break;
        default:                                          //其他情况
```



```
            cout << "input error" << endl;
        }
}
```



Note

程序需要用户输入一个字符，当用户输入字符 A 时，向屏幕输出“very good”字符串；输入字符 B 时，向屏幕输出“good”字符串；输入字符 C 时，向屏幕输出“normal”字符串；输入字符 D 时，向屏幕输出“failure”字符串；输入其他字符时，向屏幕输出“input error”字符串。

可以将 switch 的判断结构改为第一种形式的判断语句。

【例 5.9】 根据输入的字符输出字符串。

实例位置：光盘\MR\Instance\05\5.9

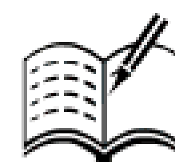
```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
    cout<<"输入一个 A-D 范围内的大写字母作为成绩评价"<<endl;
    int iInput;
    cin >> iInput;
    if(iInput = 'A')                                    //用 if 形式选择输出内容
    {
        cout << "very good" <<endl;
        return ;
    }
    if(iInput = 'B')
    {
        cout << "good" <<endl;
        return ;
    }
    if(iInput = 'C')
    {
        cout << "normal" <<endl;
        return ;
    }
    if(iInput = 'D')
    {
        cout << "failure" <<endl;
        return ;
    }
    cout << "input error" << endl;
}
```

实例 5.9 和实例 5.8 完成的功能基本相同。当用户输入字符 A 后，输出字符串“very good”，所不同的是，输出完字符串后，使用 return 跳出主函数，并结束程序，不执行下面的语句。同样输入字符 B、C 和 D 后也输出对应的字符串后跳出主函数并结束程序。

也可以将 switch 的判断结构改为第三种形式的判断语句。

【例 5.10】 根据输入的字符输出字符串。





实例位置：光盘\MR\Instance\05\5.10

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	cout<<"输入一个 A-D 范围内的大写字母作为成绩评价"<<endl;
	char iInput;
	cin >> iInput;
	if(iInput == 'A')
	{
		cout << "very good" <<endl;
		return ;
	}else if(iInput == 'B')
	{
		cout << "good" <<endl;
		return ;
	}else if(iInput == 'C')
	{
		cout << "normal" <<endl;
		return ;
	}else if(iInput == 'D')
	{
		cout << "failure" <<endl;
		return ;
	}else
		cout << "input error" << endl;
}
```

同样，本程序也是根据输入不同的字符输出不同的字符串。

switch 语句中每个 case 语句都使用 break 语句跳出，该语句可以省略。由于程序默认为顺序执行，当语句匹配成功后，其后面的每条 case 语句都会被执行，而不进行判断。

【例 5.11】 不加 break 的 switch 判断语句。

实例位置：光盘\MR\Instance\05\5.11

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	cout<<"输入一个 1-7 范围内的数字作为相应的星期"<<endl;
	int iInput;
	cin >> iInput;
	switch(iInput)
	{
		case 1:
			cout << "Monday" << endl;
		case 2:
```





Note

```
                cout << "Tuesday" << endl;
            case 3:
                cout << "Wednesday" << endl;
            case 4:
                cout << "Thursday" << endl;
            case 5:
                cout << "Friday" << endl;
            case 6:
                cout << "Saturday" << endl;
            case 7:
                cout << "Sunday" << endl;
            default:
                cout << "Input error" << endl;
        }
}
```

当输入 1 时，程序运行结果如图 5.8 所示。



图 5.8　运行结果 1

当输入 7 时，程序运行结果如图 5.9 所示。



图 5.9　运行结果 2

程序想要实现根据输入的 1～7 中的任意整型数，输出整型数对应的英文星期名称，但由于 switch 语句中的各 case 分句没有及时使用 break 语句跳出，导致意想不到的结果输出。





Note

5.5　判断语句的嵌套

前面讲过 3 种形式的判断语句，这 3 种判断语句都可以嵌套判断语句。例如，在第一种形式的判断语句中嵌套第二种形式的判断语句，形式如下：

```
if(表达式 1)
{
        if(表达式 2)
语句 1;
else
        语句 2;
}
```

在第二种形式的判断语句中嵌套第一种形式的判断语句，形式如下：

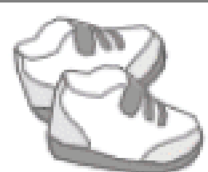
```
if(表达式 1)
{
        if(表达式 2)
                语句 1;
        else
                语句 2;
}
else
{
        if(表达式 2)
                语句 1;
        else
                语句 2;
}
```

判断语句可以有多种嵌套方式，可以根据具体需要进行设计，但一定要注意逻辑关系的正确处理。

【例 5.12】 判断是否是闰年。

实例位置：光盘\MR\Instance\05\5.12

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main()
{
        int iYear;
        cout << "请输入年份" << endl;
        cin >> iYear;
        if(iYear%4==0)                          //被 4 整除
        {
```



```
        if(iYear%100==0)                        //被 100 整除
        {
            if(iYear%400==0)                    //被 400 整除
                cout << "这是个闰年" << endl;
            else
                cout << "这不是个闰年" << endl;
        }
        else
            cout << "这是个闰年" << endl;       //提示是闰年
    }
    else
        cout << "这不是个闰年" << endl;         //提示不是闰年
    return 0;
}
```



Note

判断闰年的方法是看该年份能否被 4 整除、不能被 100 整除但能被 400 整除。程序使用判断语句对这 3 个条件逐一判断，先判断年份能否被 4 整除 iYear%4==0，如果不能整除，输出字符串“这不是个闰年”，如果能整除，继续判断能否被 100 整除 iYear%100==0，如果不能整除，输出字符串“这是个闰年”，如果能整除，继续判断能否被 400 整除 iYear%400==0，如果能整除输出字符串“这是个闰年”，不能整除输出字符串“这不是个闰年”。

可以简化判断是否是闰年的实例代码，用一条判断语句来完成。

【例 5.13】 判断是否是闰年。

实例位置：光盘\MR\Instance\05\5.13

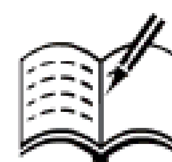
```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
    int iYear;
    cout << "请输入年份" << endl;
    cin >> iYear;                                   //输入年份
    if(iYear%4==0 && iYear%100!=0 || iYear%400==0)  //能被 4 和 100 整除或者能被 400 整除
        cout << "这是个闰年" << endl;               //是闰年
    else
        cout << "这不是个闰年" << endl;             //非闰年
}
```

程序中将能否被 4 整除、不能被 100 整除但能被 400 整除这 3 个条件用一个表达式来完成。表达式是一个复合表达式，进行了 3 次算术运算和两次逻辑运算，算术运算判断能否被整除，逻辑运算判断是否满足 3 个条件。

使用判断语句嵌套时要注意 else 关键字要和 if 关键字成对出现，并且遵守临近原则，else 关键字和自己最近的 if 语句构成一对。另外，判断语句应尽量使用复合语句，以免产生二义性，导致书写格式的运行结果和设计时的不一致。

程序中也可以出现多个独立的 if 与 else 语句，形式如下：






```
if(表达式 1)
{
    语句 1;
    …;
}
if(表达式 2)
{
        语句 2;
        …;
else if (表达式 3)
{
    语句 3;
        …;
}
else   //将与最近的 if 组成 if-else 判断语句
{
    语句 4;
    …;
}
```

程序会按照“就近结合”的原则，相邻的 if 关键字与 else 关键字形成一套 if-else 语句。

5.6　综 合 应 用

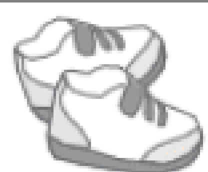
5.6.1　图书的位置

【例 5.14】 图书编号由两位数字组成，第一位代表类别，数字 1 是文学类书籍、数字 2 代表的是社科类书籍、数字 3 代表的是历史类书籍、数字 4 代表的是人物传记。第二位数字代表此书位于书架的第几层。这个图书馆的书架按类别分开，每个书架都有 4 层。设计一个程序，输入一个编号后，输出书应该位于哪个书架的第几层，当编号无效时，应给予提示并再次输入。

提示：程序应先判断输入的编号是否有效。当输入的数字被判断为无效，应使用一个循环回到这个输入语句。若编号有效，由第一位数字的值判断书籍在哪一类图书的书架上，由后一位数字判断书籍所处书架的层数。代码如下：

实例位置：光盘\MR\Instance\05\5.14

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main(int argc, _TCHAR* argv[])
{
    do{
        int num,kind,row;
        cout<<"输入一个两位的图书编号"<<endl;
        cin>>num;
```





Note

```
            kind = num/10;
            row = num%10;
            if(kind<1||kind>4||row>4||row<1)
            {
                cout<<"您输入的有误"<<endl;
                continue;                              //寻找循环判断条件，这里是 while
            }
            cout<<"此书位于";
            switch(kind)
            {
                case 1:
                    cout<<"文学类书架";
                    break;
                case 2:
                    cout<<"社科类书架";
                    break;
                case 3:
                    cout<<"历史类书架";
                    break;
                case 4:
                    cout<<"人物传记书架";
                    break;
            }
            cout<<"第"<<row<<"层"<<endl;
            break;                                     //找到位置，跳出 while
        }while(true);
        return 0;
}
```

程序运行效果如图 5.10 所示。

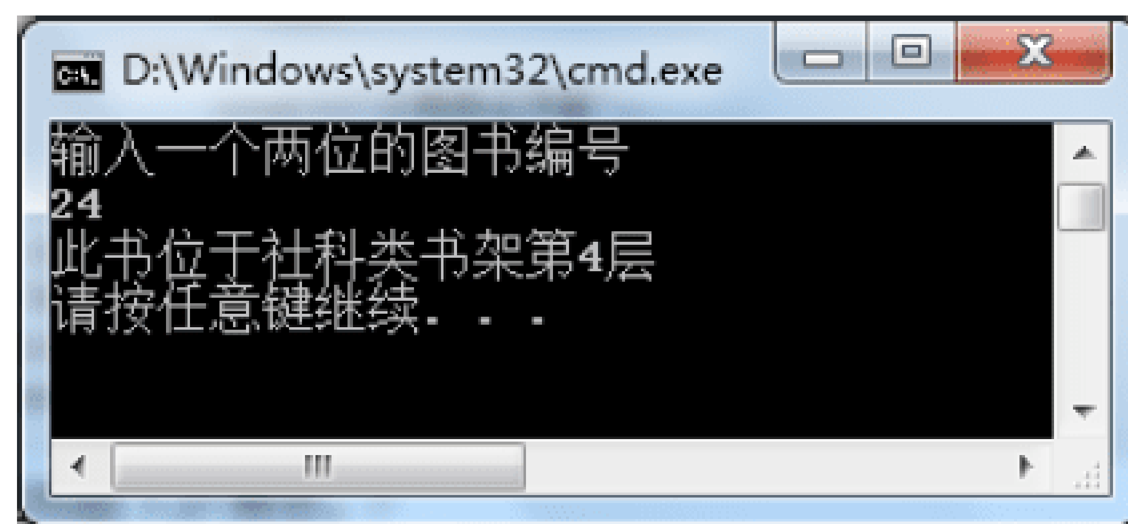


图 5.10　图书的位置

5.6.2　计算增加后的工资

【例 5.15】 编写一个程序，计算增加后的工资。

要求：基本工资大于或等于 5000 元，增加 10%工资；若小于 2500 元，且大于或等于 5000 元，则增加 15%；若小于 2500 元，则增加 20%工资。代码如下：

实例位置：光盘\MR\Instance\05\5.15

```
#include "stdafx.h"
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
int _tmain(int argc, _TCHAR* argv[])
{
        float laborage;
```





```
        cout<<"please input your laborage:\n";
        cin>>laborage;
        if(laborage>=5000)
                cout<<"Now the laborage is "<<laborage*1.1<<endl;
        if(laborage>=2500&&laborage<5000)
                cout<<"Now the laborage is "<<laborage*1.15<<endl;
        if(laborage<2500)
                cout<<"Now the laborage is "<<laborage*1.2<<endl;
        return 0;
}
```

程序运行结果如图 5.11 所示。

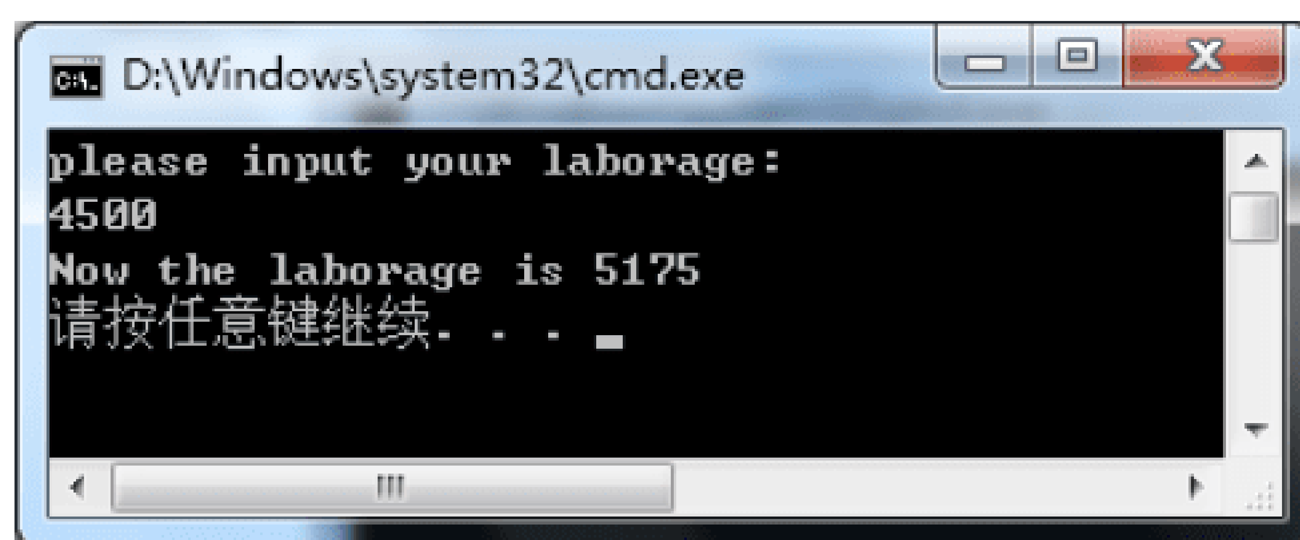


图 5.11　计算增加后的工资

5.7　本章常见错误

5.7.1　注意 case 后不要跟变量

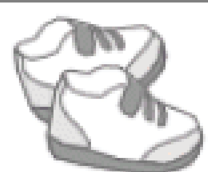
使用 switch 判断语句时提示“error C2051: case expression not constant”。

这是 case 后面的选项不正确导致的。case 后面要接常量或常量表达式，以执行相应的操作。不能用变量。例如：

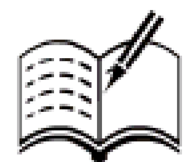
```
int a = 1;                    //变量
int n = 0;
switch(n)
{
        case a:               //错误
        case 1:               //正确
}
```

注意，在 C 语言中，case 后面只能用常量，在 C++中，case 后面可以是常量也可以是只读变量（用 const 修饰）。例如：

```
//cpp 中
const int a = 1;              //声明只读变量 a
switch(n)
{
```



```
    case a:                  //正确
}
```



Note

5.7.2 if else 的匹配问题

else 语句和它前面离它最近的一个 if 相匹配。当有多个 if else 嵌套时，很容易混乱。最好的办法是严格按照规范的代码格式编写代码，注意代码的缩进与对齐，这样可以更直观地看出匹配关系。

5.7.3 if 判断表达式的比较问题

整型值要和整型值作比较，指针要和 NULL 比较。浮点型数值不是准确值，最好不作等值比较。

5.8 本章小结

本章主要讲解了 C++语句中各种形式的判断语句，每种形式的语句都可以用另外一种格式代替，这增加了开发程序的灵活性。如果是简单的判断建议用条件运算符，如果分支较多的逻辑判断，建议使用 switch 语句，还要特别注意判断语句的书写格式，避免产生二义性。

5.9 跟我上机

参考答案：光盘\MR\跟我上机

开发一个程序，要求在输入 1 时，显示星期一；输入 2 时，显示星期二；依此类推，输入 0 时，显示星期天，输入 7 退出程序。程序代码如下：

```
#include "stdafx.h"
#include <iostream>
using std::cin;
using std::cout;
using std::endl;
int _tmain(int argc, _TCHAR* argv[])
{
    int week;
    cout << "请输入(0-6)的数字：(按 7 退出)\n";
    while(1)
    {
```





```
            cin >> week;
            switch (week)
            {
            case 0:
                cout << "星期日";
                break;
            case 1:
                cout << "星期一";
                break;
            case 2:
                cout << "星期二";
                break;
            case 3:
                cout << "星期三";
                break;
            case 4:
                cout << "星期四";
                break;
            case 5:
                cout << "星期五";
                break;
            case 6:
                cout << "星期六";
                break;
            case 7:
                return 0;
            default:
                cout << "输入错误，重新输入\n";
            }
            cout << '\n';
        }
        return 0;
}
```

第6章

循环控制语句

（视频讲解：50 分钟）

循环控制就是控制程序重复执行，当不符合循环条件时停止循环。使用循环结构可以使循环代码更加简洁，减少冗余。掌握循环结构是程序设计的最基本要求，本章主要介绍了 while 循环、do…while 循环和 for 循环语句，这 3 种循环语句可以互相转换，达到同一目标可以运用多种方法。

本章能够完成的主要范例（已掌握的在方框中打勾）

- [ ] 了解 3 种循环语句
- [ ] 掌握各种循环的区别
- [ ] 了解循环的跳转
- [ ] 掌握循环的嵌套





6.1　while 循环

while 循环语句的一般形式如下：

```
while(表达式)
{
    重复执行的内容
}
```

表达式一般是一个关系表达式或逻辑表达式，其表达式的值应该是一个逻辑值（true 和 false），当表达式的值为真时开始循环执行语句，当值为假时退出循环，执行循环外的下一条语句。循环每次都是执行完语句后回到表达式处重新开始判断并计算表达式的值，一旦值为假时就退出循环，为真时就继续执行语句。while 循环可以用流程来演示执行过程，如图 6.1 所示。

语句可以是复合语句，也就是用花括号括起多条简单语句。花括号及其所包括的语句，被称为循环体，循环主要指循环执行循环体的内容。

【例 6.1】 使用 while 循环计算从 1 到 10 的累加。

实例位置：光盘\MR\Instance\06\6.1

1 到 10 的累加就是计算 1+2+…+10，需要有一个变量从 1 变化到 10，将该变量命名为 i，还需要另外一个临时变量不断和该变量进行加法运算，并记录运算结果，将临时变量命名为 sum，变量 i 每增加 1 时，就和变量 sum 进行一次加法运算，变量 sum 记录的是累加的结果。程序需要使用循环语句，使用 while 循环需要将循环语句的结束条件设置为 i<=10，循环流程如图 6.2 所示。

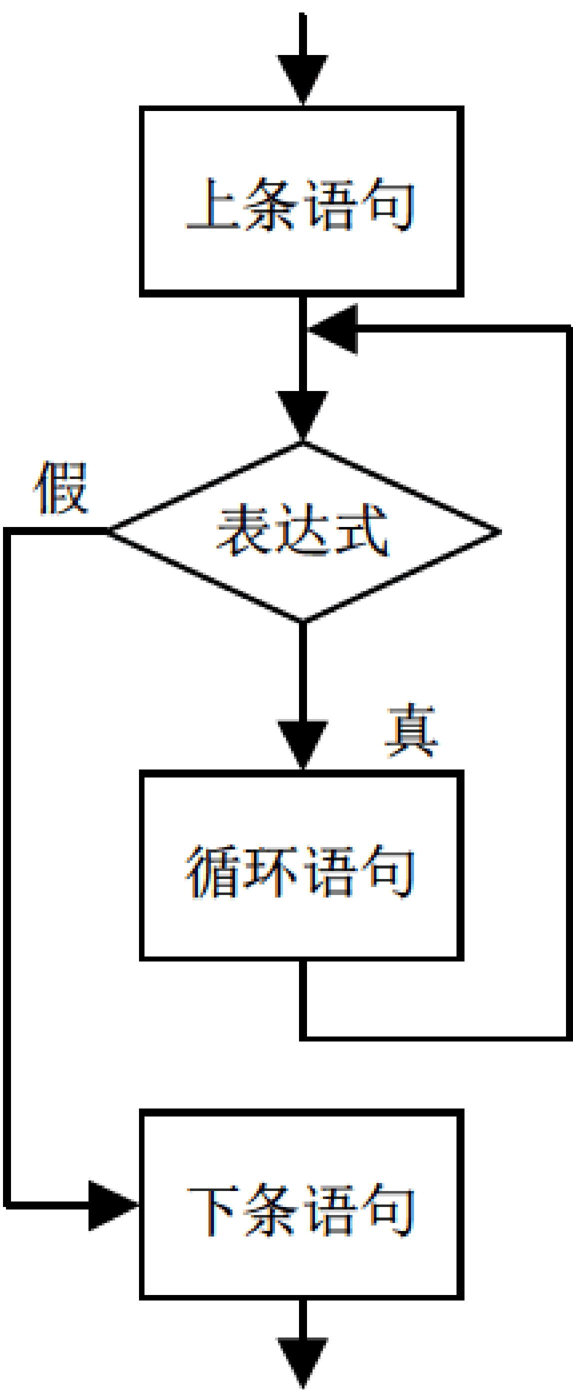


图 6.1　while 循环

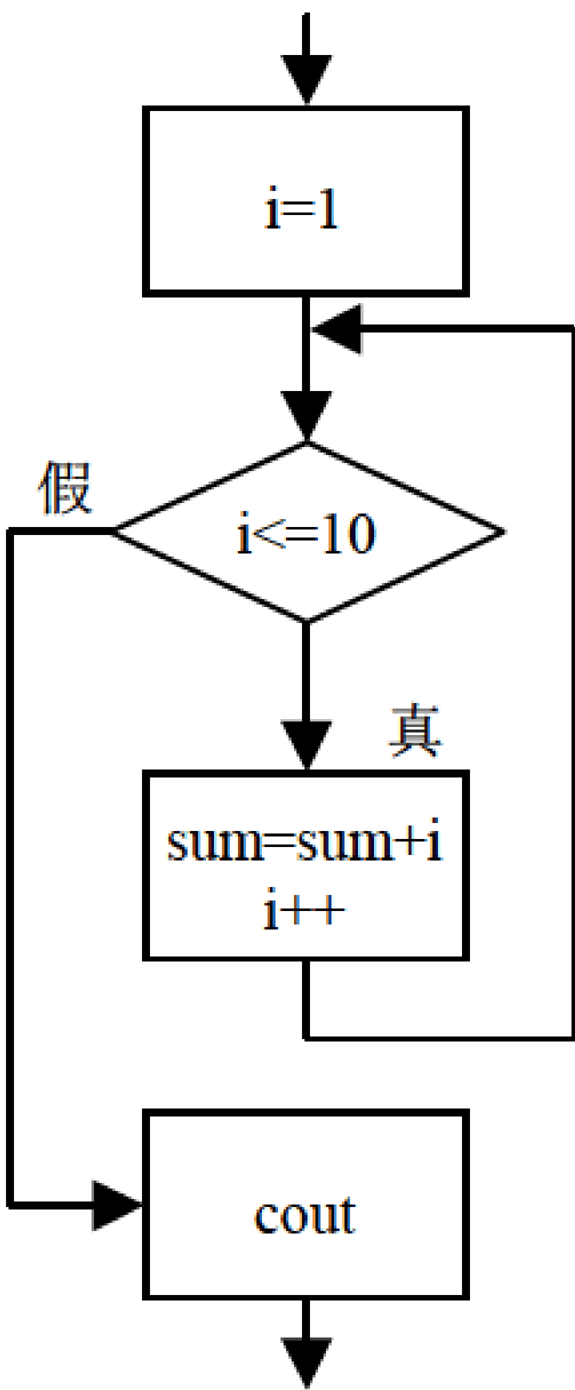


图 6.2　while 循环计算从 1 到 10 的累加



程序代码如下：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int sum=0,i=1;
	while(i<=10)
	{
		sum=sum+i;
		i++;
	}
	cout << "数字 1-10 之和:" << sum << endl;
}
```

程序运行结果如图 6.3 所示。

程序先对变量 sum 和 i 进行初始化，while 循环语句的表示式是 i<=10，所要执行的循环体是一个复合语句，是由“sum=sum+i;”和“i++;”两条简单语句完成，语句“sum=sum+i;”完成累加，语句“i++;”完成由 1 到 10 的递增变化。



图 6.3　程序运行结果



注意：

使用 while 循环的注意事项如下。

（1）表达式不可以为空，表达式为空不合法。

（2）表达式可以用非 0 代表逻辑值真（true），用 0 代表逻辑值假（false）。

（3）循环体中必须有改变条件表达式值的语句，否则将成为死循环。例如：

```
while(1)    //也可以写作 while(true)
{
…
}
```

是一个无限循环语句。

例如：

```
while(0)    //也可以写作 while(false)
{
…
}
```

是一个不会进行循环的语句。





6.2　do…while 循环

do…while 循环语句的一般形式如下：

```
do
语句
while(表达式)
```

do 为关键字，必须与 while 配对使用。do 与 while 之间的语句称为循环体，该语句同样是用大括号{}括起来的复合语句。循环语句中的表达式与 while 语句中的相同，也多为关系表达式或逻辑表达式。但特别值得注意的是 do…while 语句后要有分号“;”。do…while 循环可以用流程来演示执行过程，如图 6.4 所示。

do…while 循环的执行顺序是执行循环体的内容，然后判断表达式的值，如果表达式的值为真就跳到循环体处继续执行循环体，循环一直到表达式的值为假时跳出循环，执行下一条语句。

【例 6.2】 使用 do…while 循环计算 1 到 10 的累加。

实例位置：光盘\MR\Instance\06\6.2

1 到 10 的累加就是计算 1+2+…+10 的值，前面的例子使用 while 循环语句实现了 1 到 10 的累加，do…while 循环和 while 循环实现累加的循环体语句相同，只是执行循环体的先后顺序不同，程序执行顺序如图 6.5 所示。

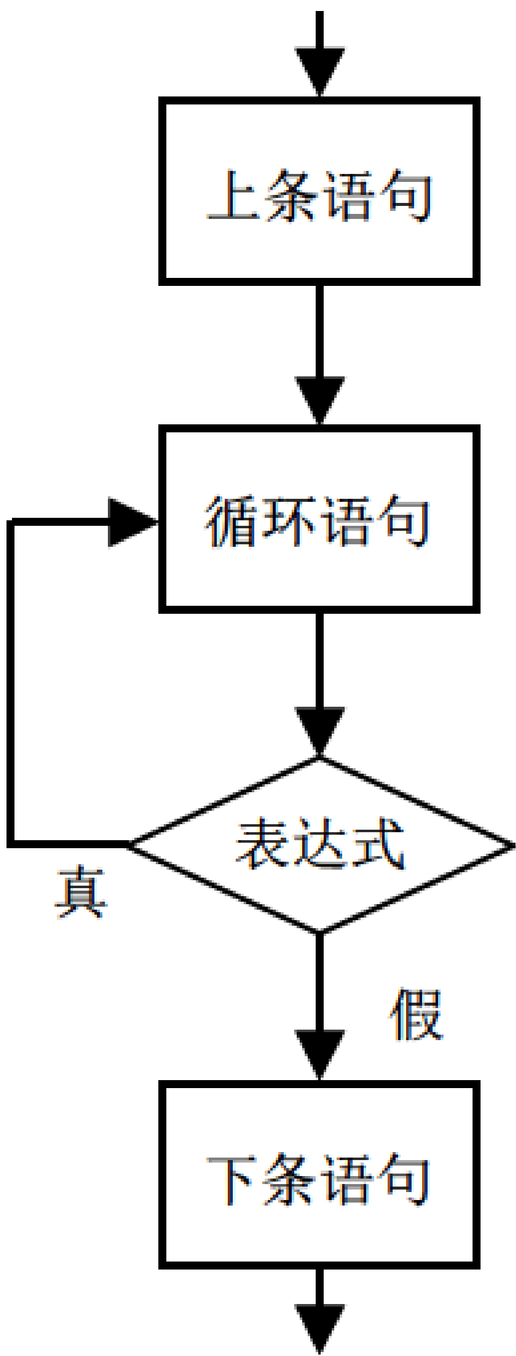


图 6.4　do…while 循环

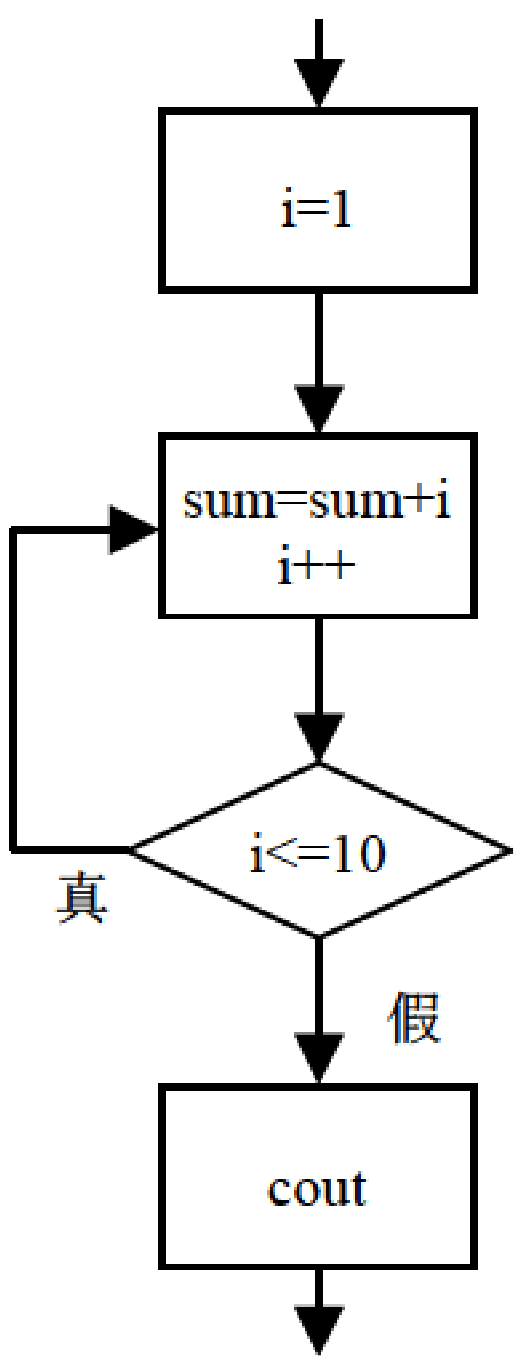
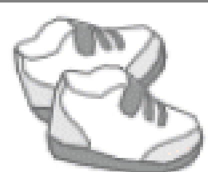


图 6.5　do…while 循环计算 1 到 10 的累加

程序代码如下：

```
#include "stdafx.h"
#include <iostream>
```





```
using namespace std;
void main()
{
	int sum=0,i=1;
	do
	{
		sum=sum+i;
		i++;
	}while(i<=10);
	cout << "数字 1-10 之和 :" << sum << endl;
}
```

程序运行结果如图 6.6 所示。



图 6.6　程序运行结果

程序使用变量 sum 作为记录累加的结果，变量 i 完成由 1 到 10 的变化，程序先将变量 sum 初始化为 0，将变量 i 初始化为 1，先执行循环体变量 sum 和变量 i 的加法运算，并将运算结果保存到变量 sum，然后变量 i 进行自加运算，接着判断循环条件，看变量 i 的值是否已经大于 10 了，如果大于 10 就跳出循环，小于或等于 10 就继续执行循环体语句。



注意：

do…while 循环的注意事项：

（1）循环先执行循环体，如果循环条件不成立，循环体已经执行一次了，使用时注意变量变化。

（2）表达式不可以为空，表达式为空不合法。

（3）表达式可以用非 0 代表逻辑值真（true），用 0 代表逻辑值假（false）。

（4）循环体中必须有改变条件表达式值的语句，否则将成为死循环。

（5）注意循环语句后要有分号“;”。

6.3　while 和 do…while 比较

可以通过设置起始循环条件不成立循环语句，来观察 while 和 do…while 的不同。将变量 i 初始值设置为 0，然后循环表达式设置为 i>1，显然循环条件不成立。循环体执行的是对变量 j 的加 1 运算，通过输出变量 j 在循环前的值和循环后的值来进行比较。

【例 6.3】 使用 do…while 循环进行计算。

实例位置：光盘\MR\Instance\06\6.3

使用 do…while 循环进行计算，代码如下：






```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int i=0,j=0;
	cout << "before do_while j=" << j << endl;
	do
	{
		j++;
	}while(i>1);
	cout << " after do_while j=" << j << endl;
}
```

程序运行结果如图 6.7 所示。

【例 6.4】 使用 while 循环进行计算。

实例位置：光盘\MR\Instance\06\6.4

使用 while 循环进行计算，代码如下：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int i=0,j=0;
	cout << "执行 while 前 j=" << j << endl;
	while(i>1)
	{
		j++;
	}
	cout << "执行 while 后 j=" << j << endl;
}
```

程序运行结果如图 6.8 所示。

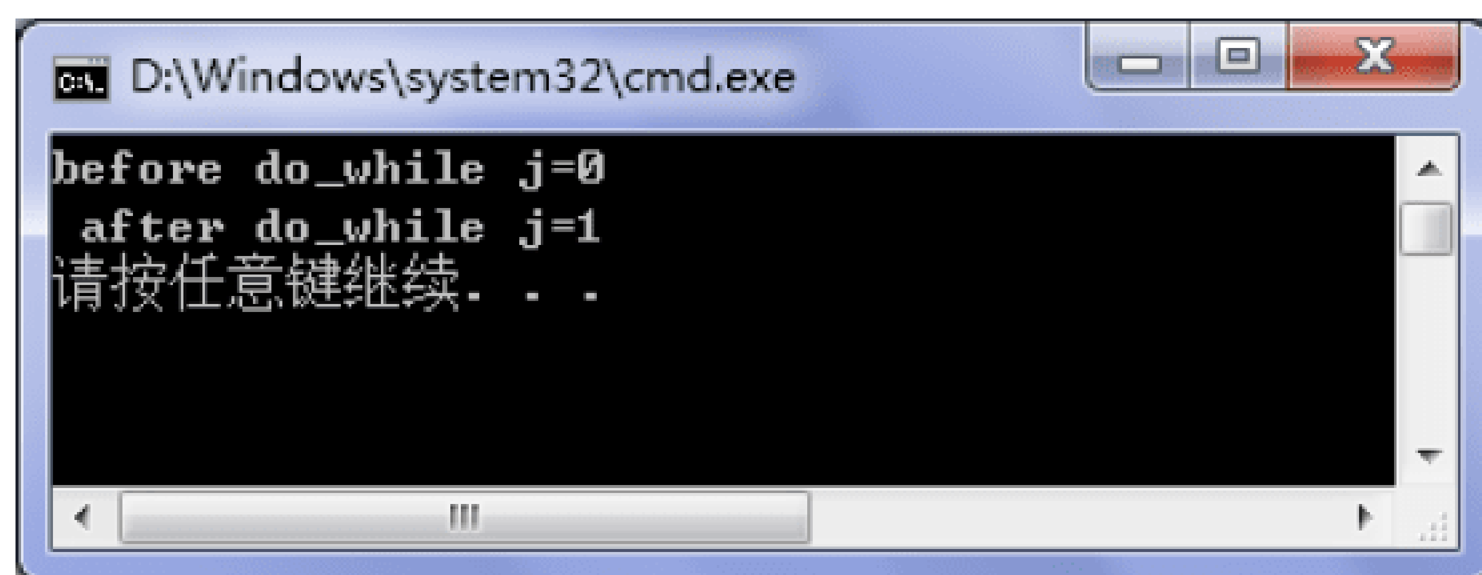


图 6.7　do…while 循环

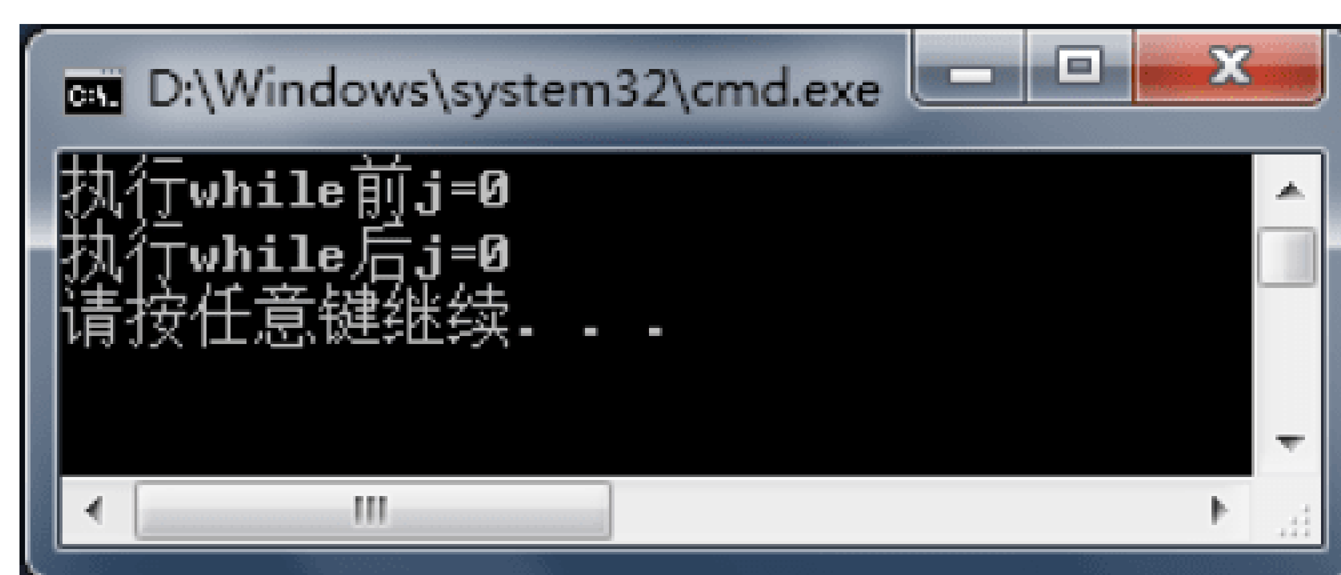
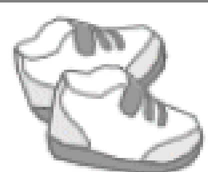


图 6.8　while 循环

使用 do…while 循环后变量 j 的值为 1，而使用 while 循环后变量 j 的值仍为 0。





Note

6.4 for 循环

for 循环语句的一般格式如下：

```
for(表达式 1;表达式 2;表达式 3)语句
```

- ☑ 表达式 1：该表达式通常是一个赋值表达式，负责设置循环的起始值，也就是给控制循环的变量赋初值。
- ☑ 表达式 2：该表达式通常是一个关系表达式，用控制循环的变量和循环变量允许的范围值进行比较。
- ☑ 表达式 3：该表达式通常是一个赋值表达式，对控制循环的变量进行增大或减小。
- ☑ 语句：是复合语句。

for 循环语句的执行过程如下：

（1）先求解表达式 1。

（2）求解表达式 2，若其值为真，则执行 for 语句中指定的内嵌语句，然后执行（3）。若表达式 2 值为 0，则结束循环，转到（5）。

（3）求解表达式 3。

（4）返回（2）继续执行。

（5）循环结束，执行 for 语句下面的一个语句。

上面的 5 个步骤也可以用图 6.9 表示。

【例 6.5】 用 for 循环计算从 1 到 10 的累加。

实例位置：光盘\MR\Instance\06\6.5

for 循环不同于 while 和 do…while 循环，它有 3 个表达式，需要正确设置这 3 个表达式。计算累加需要一个能由 1 到 10 递增变化的变量 i 和一个记录累加和的变量 sum，for 循环的表达式中可以对变量进行初始化，以及实现变量由 1 到 10 的递增变化。循环执行顺序如图 6.10 所示。

程序代码如下：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
        int sum=0;
        int i;
        for(i=1;i<=10;i++)                    //for 循环语句
            sum+=i;
        cout << "数字 1-10 的和:" << sum << endl;
}
```

程序运行结果如图 6.11 所示。





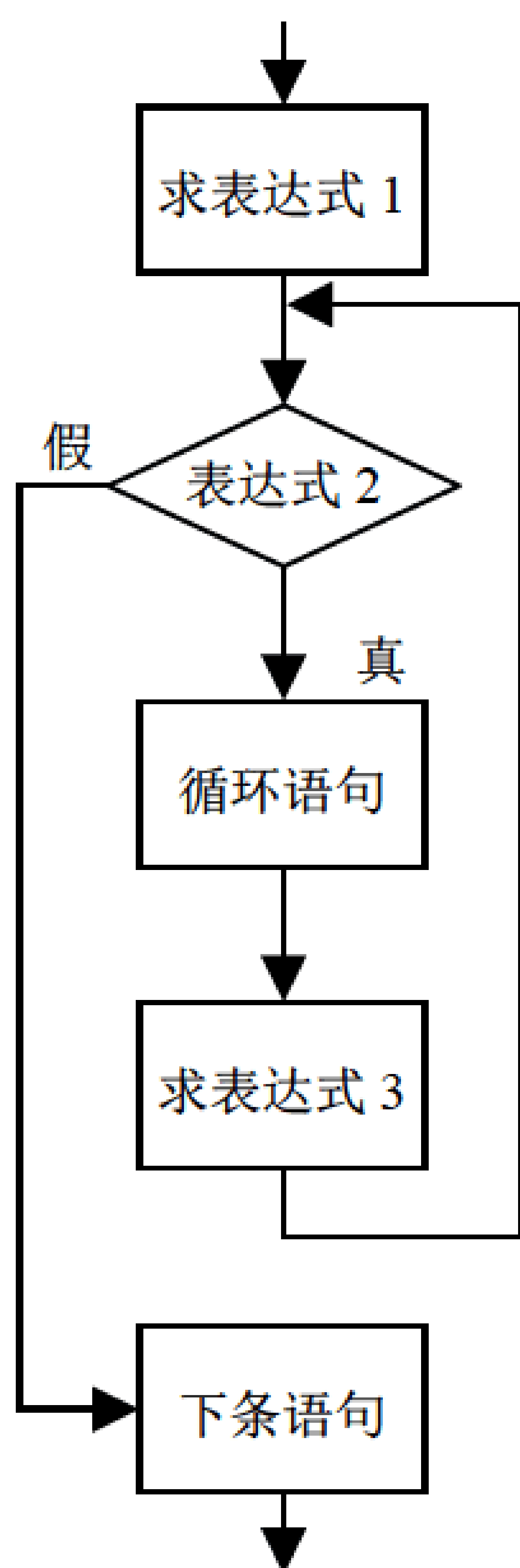


图 6.9　for 循环执行过程

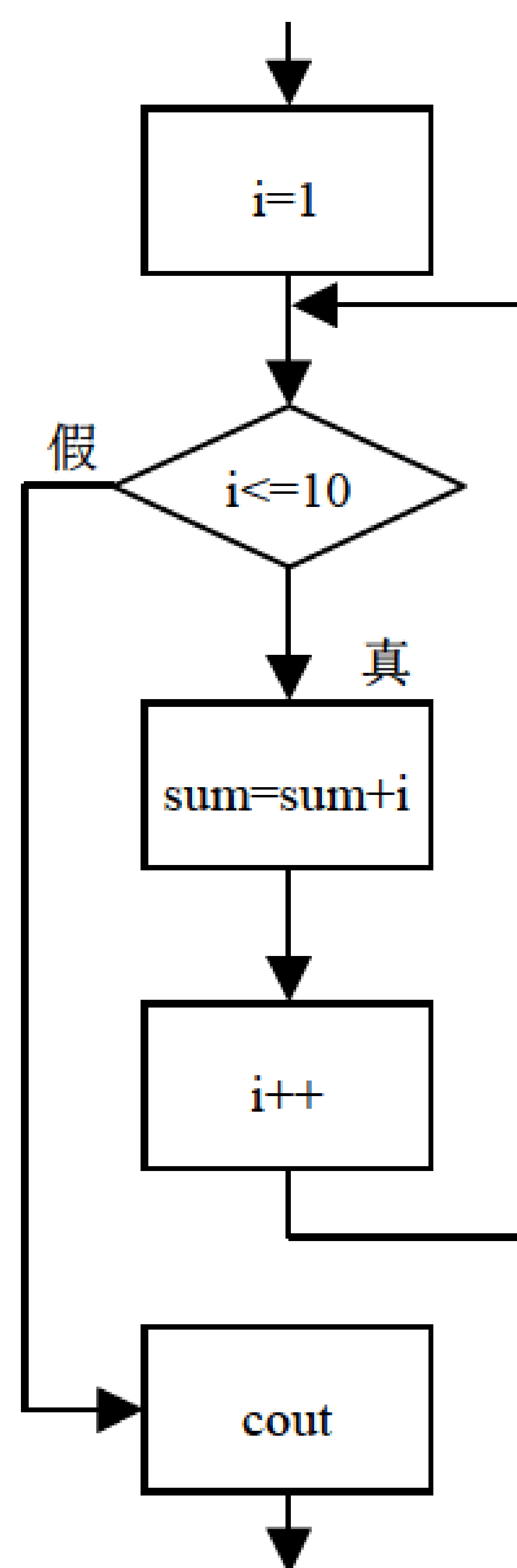


图 6.10　for 循环执行顺序



图 6.11　运行结果

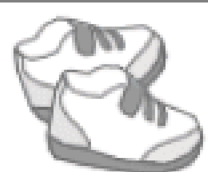
程序中“for(i=1;i<=10;i++) sum+=i;”就是一个循环语句，sum+=i 是循环体语句，其中 i 就是控制循环的变量，i=1 是表达式 1，i<=10 是表达式 2，i++是表达式 3，sum+=i 是语句；表达式 1 将循环控制变量 i 赋初始值为 1，表达式 2 中 10 是循环变量允许的范围，也就是说 i 不能大于 10，大于 10 时将不执行语句“sum+=i;”。语句“sum+=i;”是使用了带运算的赋值语句，它等同于语句“sum=sum+i;”。sum+=i 语句一共执行了 10 次，i 的值是从 1 到 10 变化，j+=i 语句完成 1 到 10 的累加。

for 循环的注意事项如下。

（1）for 语句可以在表达式 1 中直接声明变量。

在表达式外声明变量：

```
#include <iostream>
using namespace std;
void main()
{
	int sum=0,i;                    //在表达式外声明变量
	for(i=0;i<=10;i++)              //循环 11 次
```



```
        sum+=i;                                //0 依次加到 10
    cout <<sum << endl;
}
```



Note

在表达式内声明变量：

```
#include <iostream>
using namespace std;
void main()
{
    for(int i=0,sum=0;i<=10;i++)           //在循环语句中声明变量
        sum+=i;
    cout <<sum << endl;
}
```

在循环语句中声明变量，也相当于在函数内声明了变量，如果在表达式 1 中声明两个相同变量，编译器将报错：

```
void main()
{
    for(int i=0,sum=0;i<=10;i++)           //在循环语句中声明变量
        sum+=i;
    for(int i=0,sum=0;i<=10;i++)           //不合法，编译器报错
        sum+=i;
    cout <<sum << endl;
}
```

（2）for 循环中的表达式 1、表达式 2、表达式 3 都可以省略。

☑ 省略表达式 1

如果省略表达式 1，且控制变量在循环外声明并赋初值，程序能编译通过并且正确运行。例如：

```
#include <iostream>
using namespace std;
void main()
{
    int sum=0;
    int i=0;                               //将循环控制变量拿到循环语句外声明并赋初值
    for(;i<=10;i++)                        //循环 11 次
        sum+=i;
    cout <<sum << endl;
}
```

程序仍是计算从 1 到 10 的累加的。

如果控制变量在循环外声明但没有赋初值，程序能编译通过，但运行结果不是用户所期待的。因为编译器会为变量赋一个默认的初值，该初值一般为一个比较大的负数，所以会造成运行结果不正确。





☑　省略表达式 2

省略了表达 2 也就是省略了循环判断语句，没有循环的终止条件，循环变成无限循环。

☑　省略表达式 3

省略表达式 3 后循环也是无限循环，因为控制循环的变量永远都是初始值，永远符合循环条件。

☑　省略表达式 1 和表达式 3

for 循环语句如果省略表达式 1 和表达式 3，就和 while 循环一样了。例如：

```
#include <iostream>
using namespace std;
void main()
{
	int sum=0;
	int i=0;
	for(;i<=10;)
	{
		sum=sum+i;
		i++;
	}
	cout << "the result :" << sum << endl;
}
```

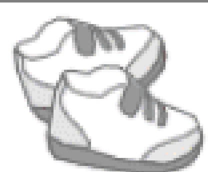
☑　3 个表达式同时省略

for 循环语句如果省略 3 个表达式，就会变成无限循环。无限循环就是死循环，它会使程序进入瘫痪状态。使用循环时，建议使用计数控制，也就是说循环执行到指定次数，就跳出循环。例如：

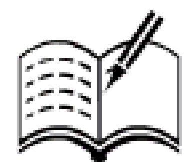
```
void main()
{
	int iCount=0;                    //声明用于计数的变量
	for(;;)                          //都省略，这里不设循环条件
	{
		…
		iCount++;                    //每循环一次，计数器加 1
		if(iCount>200000)            //如果循环次数大于 200000 跳出循环
			return;                  //结束循环
	}
	cout << "the loop end" << endl;
}
```

6.5　循环控制

循环控制包含两方面的内容，一方面是控制循环变量的变化方式，一方面是控制循环的跳转。



控制循环的跳转需要用到 break 和 continue 两个关键字，这两条跳转语句的跳转效果不同，break 是中断循环，continue 是跳出本次循环体的执行。



Note

6.5.1 控制循环的变量

无论是 for 循环还是 while、do…while 循环，都需要循环一个控制循环的变量，while、do…while 循环的控制变量变化可以是显式的也可以是隐式的。例如，在读取文件时，在 while 循环中循环读取文件内容，但程序中没有出现控制变量。代码如下：

```
#include <iostream>
#include <fstream>
using namespace std;
void main()
{
	ifstream ifile("test.dat",std::ios::binary);
	if(!ifile.fail())
	{
		while(!ifile.eof())				//判断文件是否结束
		{
			char ch;
			ifile.get(ch);				//获取文件内容
			if(!ifile.eof())			//如果是文件结束，就不进行最后输出
			std::cout << ch;
		}
	}
}
```

程序中 while 循环中的表达式是判断文件指针是否指向文件末尾，如果是，就跳出循环，起始程序中控制循环的变量是文件的指针，文件的指针在读取文件时不断变化。

for 循环的循环控制变量的变化方式有两种，一个是递增方式，另一个是递减方式。使用哪种方式和变量的初值和范围值的比较有关。

如果初值大于限定范围的值，表达式 2 是大于关系（>）判定的不等式，使用递减方式。

如果初值小于限定的范围值，表达式 2 是小于关系（<）判定的不等式，使用递增方式。

前文使用 for 循环计算 1 到 10 累加和使用的是递增方式，也可以使用递减方式计算 1 到 10 累加和。代码如下：

```
#include <iostream>
using namespace std;
void main()
{
	int sum=0;					//定义存储累加和变量
	for(int i=10;i>=1;i--)
		sum+=i;					//进行累加
	cout << "the result :"<<sum << endl;
}
```





程序中 for 循环的表达式 1 中声明变量并赋初值 10，表达式 2 中限定范围的值就是 1，不等式是循环控制变量 i 是否大于等于 1，如果小于 1 就停止循环，循环控制变量就是由 10 到 1 递减变化。程序输出结果仍是“the result :55”。

6.5.2　break 语句

使用 break 语句可以跳出 switch 结构。在循环结构中，同样也可用 break 语句跳出当前循环体，从而中断当前循环。

在 3 种循环语句中使用 break 语句的形式如图 6.12 所示。

```
while(...)        do                for
{                 {                 {
    ...               ...               ...
    break;            break;            break;
    ...               ...               ...
}                 }while(...);      }
```

图 6.12　break 语句的使用形式

【例 6.6】　使用 break 跳出循环。

实例位置：光盘\MR\Instance\06\6.6

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int i,n,sum;
	sum=0;
	cout<< "请输入 10 个整数" << endl;
	for(i=1;i<=10;i++)                    //循环 10 次
	{
		cout<< i<< ":" ;                  //输出 i 值
		cin >> n;                         //输入 n 值
		if(n<0)                           //判断输入是否为负数
			break;
		sum+=n;                           //对输入的数进行累加
	}
	cout << "  数的和  :"<< sum << endl;
}
```

程序中需要用户输入 10 个数，然后计算这 10 个数的和。当输入数为负数时，就停止循环不再进行累加，输出前面累加结果。例如，输入 4 次数字 1，最后输入数字-1，程序运行结果如图 6.13 所示。

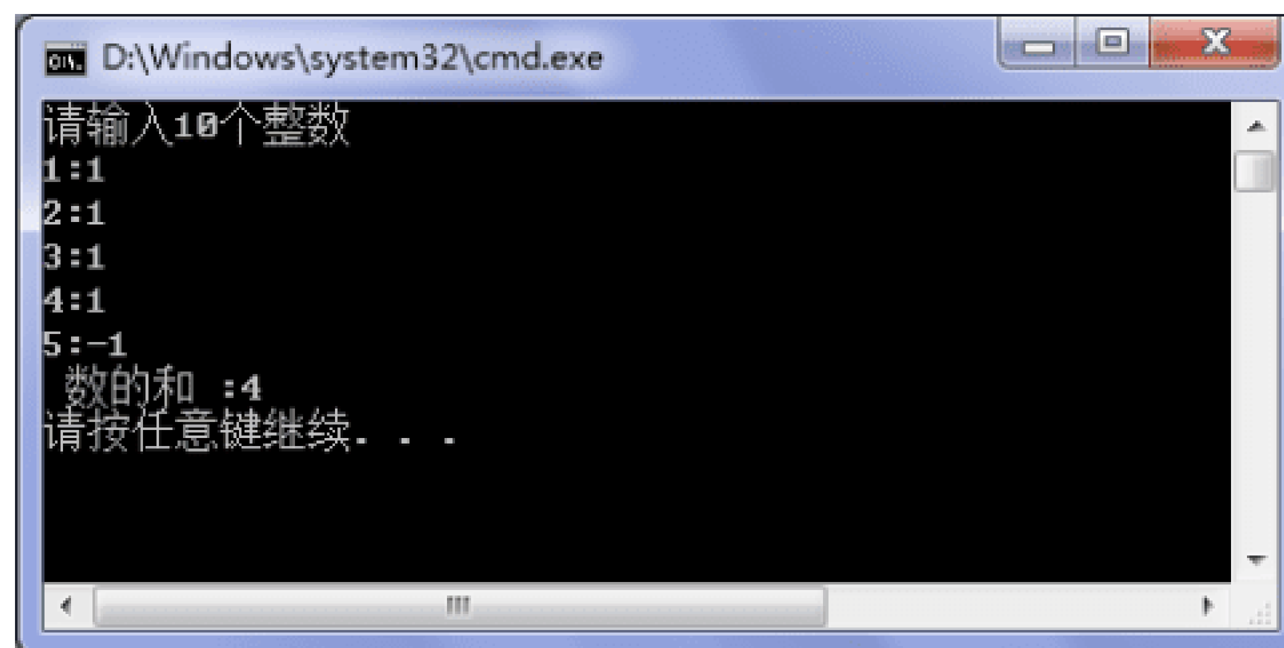
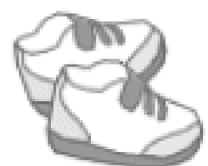
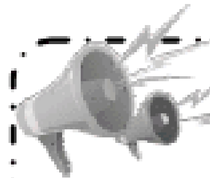


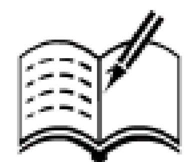
图 6.13　运行结果





注意：

如果遇到循环嵌套的情况，break 语句将只会使程序流程跳出包含它的最内层的循环结构，只跳出一层循环。



Note

6.5.3 continue 语句

continue 语句是针对 break 语句的补充。continue 不是立即跳出循环体，而是跳过本次循环结束前的语句，回到循环的条件测试部分，重新开始执行循环。在 for 循环语句中遇到 continue 后，首先执行循环的增量部分，然后进行条件测试。在 while 和 do…while 循环中，continue 语句使控制直接回到条件测试部分。

在 3 种循环语句中使用 continue 语句的形式如图 6.14 所示。

```
while(...)        do                for
{                 {                 {
    ...               ...               ...
    continue;         continue;         continue;
    ...               ...               ...
}                 }while(...);      }
```

图 6.14　continue 语句的使用形式

【例 6.7】 使用 continue 跳出循环。

实例位置：光盘\MR\Instance\06\6.7

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int i,n,sum;
	sum=0;
	cout<< "请输入 10 个整数" << endl;
	for(i=1;i<=10;i++)
	{
		cout<< i<< ":" <;
		cin >> n;
		if(n<0)			//判断输入是否为负数
			continue;		//返回上面，结束本次循环
		sum+=n;			//对输入的数进行累加
	}
	cout << "数的和:" <<sum << endl;
}
```

程序中需要用户输入 10 个数，然后计算这 10 个数的和。当输出数为负数时，不执行“sum+=n;”语句，也就是不对负数进行累加。与 break 不同的是执行完 continue 后，程序回到 for 循环处继续执行，执行结果如图 6.15 所示。





Note

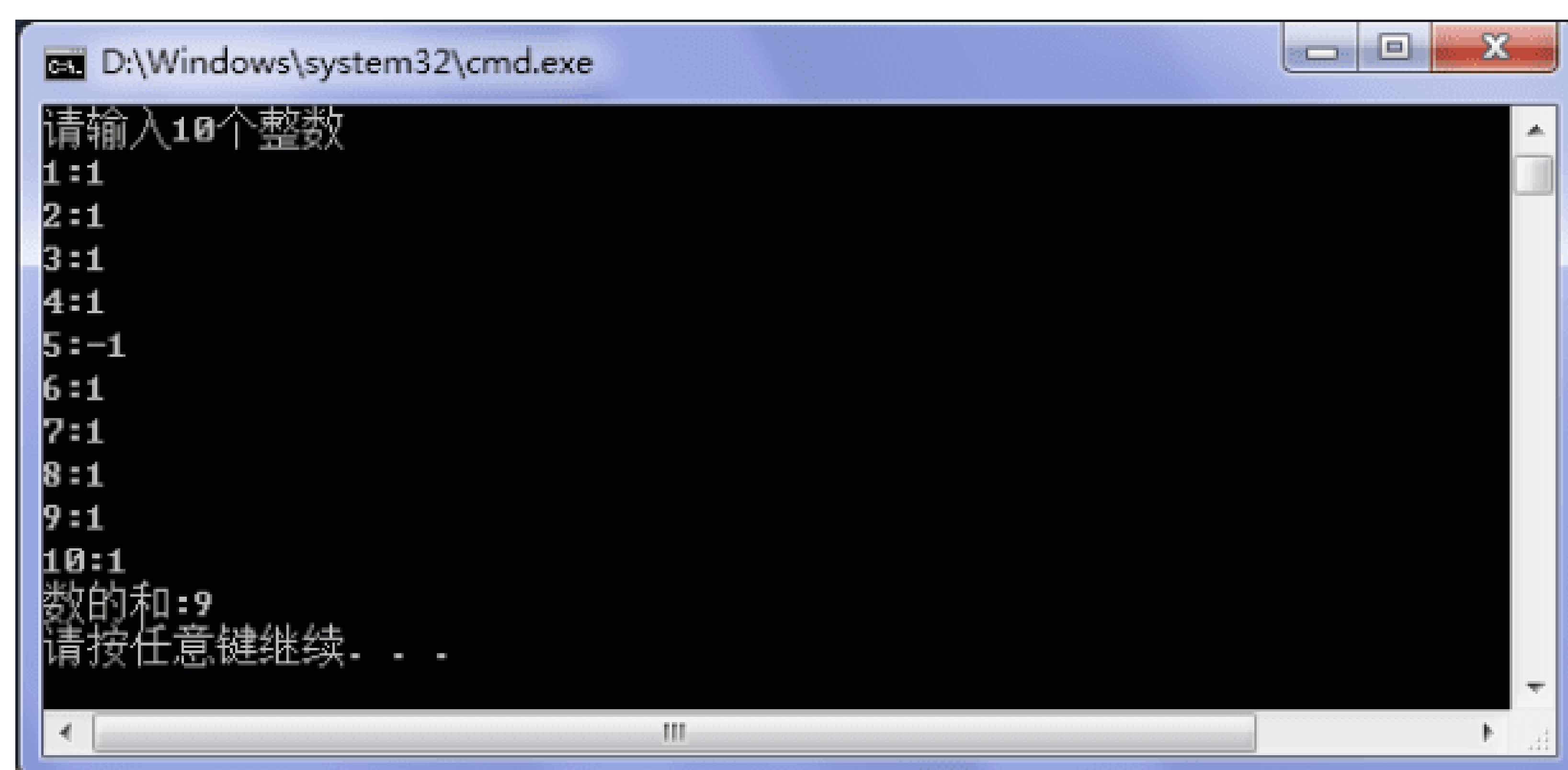


图 6.15　运行结果

6.5.4　goto 语句

goto 语句又称为无条件跳转语句，用于改变语句的执行顺序。goto 语句的一般格式为：

```
goto 标号;
```

其中，标号是用户自定义的一个标识符，以冒号结束。下面利用 goto 语句实现 1 到 100 的累加求和。

【例 6.8】 使用 goto 语句实现循环。

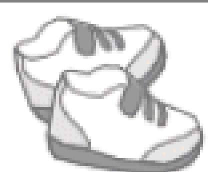
实例位置：光盘\MR\Instance\06\6.8

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int ivar = 0 ;				//定义一个整型变量，初始化为 0
	int num = 0;				//定义一个整型变量，初始化为 0
label:						//定义一个标签
	ivar++;					//ivar 自加 1
	num += ivar;				//累加求和
	if (ivar <10)				//判断 ivar 是否小于 10
	{
		goto label;			//转向标签
	}
	cout << num << endl;
}
```

执行结果：

```
55
```

程序中利用标签实现循环功能。当语句执行到 if (ivar<10)时，如果条件为真，跳转到标签定义 label:处。这是一种古老的跳转语句，它会使程序的执行顺序变得混乱，CPU 需要不停地进行跳转，效率比较低，因此，在开发程序时慎用 goto 语句。





使用 goto 语句时的说明：

（1）使用 goto 语句时，应注意标签的定义。在定义标签时，其后不能紧接着出现符号}。例如，下面的代码是非法的。

```
int ivar = 0 ;                          //定义一个整型变量，初始化为 0
int num = 0;                            //定义一个整型变量，初始化为 0
{
        …                               //其他操作
        label:                          //定义一个标签
}
```

在上述代码中定义标签时，其后没有执行代码了，所以出现编译错误。如果程序中出现上述情况，可以在标签后添加一个语句，以解决编译错误。

（2）在使用 goto 语句时还应注意 goto 语句不能越过复合语句之外的变量定义的语句。例如，下面的 goto 语句是非法的。

```
goto label;                             //跳转到标签
int i = 10;                             //声明一个变量，初始化为 10
label:                                  //定义标签
        cout<<"goto" << endll;          //输出信息
```

在上述代码中 goto 语句试图越过变量 i 的定义，导致编译错误。解决上述问题的方法是将变量的声明放在复合语句中。例如，下面的代码将是合法的。

```
goto label;                             //跳转到标签
{
        int i = 10;                     //声明一个变量，初始化为 10
}
label:                                  //定义标签
        cout<<"goto"<< endl;            //输出信息
```

6.6 循环嵌套

循环有 for、while、do…while 3 种方式，这 3 种循环可以相互嵌套。例如，在 for 循环中套用 for 循环：

```
for(…)
{
        for(…)
        {
                …
        }
}
```

在 while 循环中套用 while 循环：





```
while(…)
{
        while(…)
        {
                …
        }
}
```

在 while 循环中套用 for 循环：

```
while(…)
{
        for(…)
        {
                …
        }
}
```

【例 6.9】 打印三角形。

实例位置：光盘\MR\Instance\06\6.9

使用嵌套的 for 循环来输出由字符*组成的三角形。程序代码如下：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
    int i, j, k;
    for (i = 1; i <= 5; i++)                    //控制行数
    {
        for (j = 1; j <= 5-i; j++)              //控制空格数
            cout << " ";
        for (k = 1; k <= 2 *i - 1; k++)         //控制打印*号的数量
            cout << "*";
        cout << endl;
    }
}
```

程序中一共输出 5 行字符，最外面的 for 循环控制输出的行数，嵌套的第一个循环控制字符*前的空格数，第二个 for 循环控制输出字符*的个数。第一个循环随着行数的增加，字符*前的空格数越来越少，第二个循环输出和行号有关的奇数个字符*。程序运行结果如图 6.16 所示。

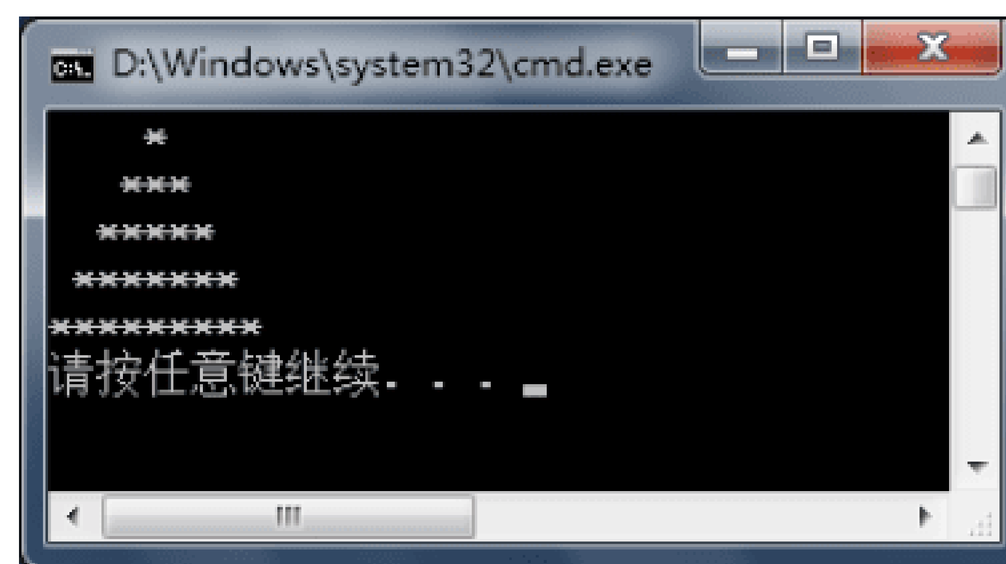


图 6.16　运行结果

【例 6.10】 按阶梯型输出九九乘法表。使用两个 for 循环嵌套，一个控制输出行数，一个控制每行输出乘式个数。实现如下：





Note

实例位置：光盘\MR\Instance\06\6.10

```
#include <iostream>
#include <iomanip>                                    //控制输出格式头文件
using std::cin;
using std::cout;
using std::endl;
int main()
{
    int Width = 0;
    for(int j=1;j<10;j++)                             //输出行数
    {
        for(int i=1;i<j+1;i++)                        //每行输出乘式个数
        {
            if(1 == i)                                //结果为 1 位数的设置 1 为宽度
                Width = 1;
            else                                      //结果为 2 位数的设置 2 为宽度
                Width = 2;
            cout<<i<<"*"<<j<<"="
                <<std::setw(Width)                    //设置输出宽度
                <<std::setiosflags(std::ios::left)    //左对齐输出
                <<i*j<<" ";                           //输出 i*j
        }
        cout<<endl;
    }
    return 0;
}
```

运行结果如图 6.17 所示。

```
D:\Windows\system32\cmd.exe
1*1=1
1*2=2 2*2=4
1*3=3 2*3=6  3*3=9
1*4=4 2*4=8  3*4=12 4*4=16
1*5=5 2*5=10 3*5=15 4*5=20 5*5=25
1*6=6 2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7=7 2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8=8 2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
1*9=9 2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81
请按任意键继续. . .
```

图 6.17 九九乘法表

6.7 综 合 应 用

6.7.1 阿姆斯壮数

【例 6.11】 实现阿姆斯壮数。

在 3 位的整数中，形如 $153=1^3+5^3+3^3$ 这样的数称为阿姆斯壮数。求阿姆斯壮数需要分别计





算出 3 位整数的百位、十位和个位，然后对 3 个数分别求立方，最后判断是否为阿姆斯壮数。

实例位置：光盘\MR\Instance\06\6.11

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main()
{
	int a,b,c;
	int input;
	for(input = 100;input<=999;input++)
	{
		a = input/100;                              //求百位
		b = (input%100)/10;                         //求十位
		c = input%10;                               //求个位
		if(a*a*a+b*b*b+c*c*c == input)
			cout<<input<<endl;
	}
	return 0;
}
```

运行效果如图 6.18 所示。

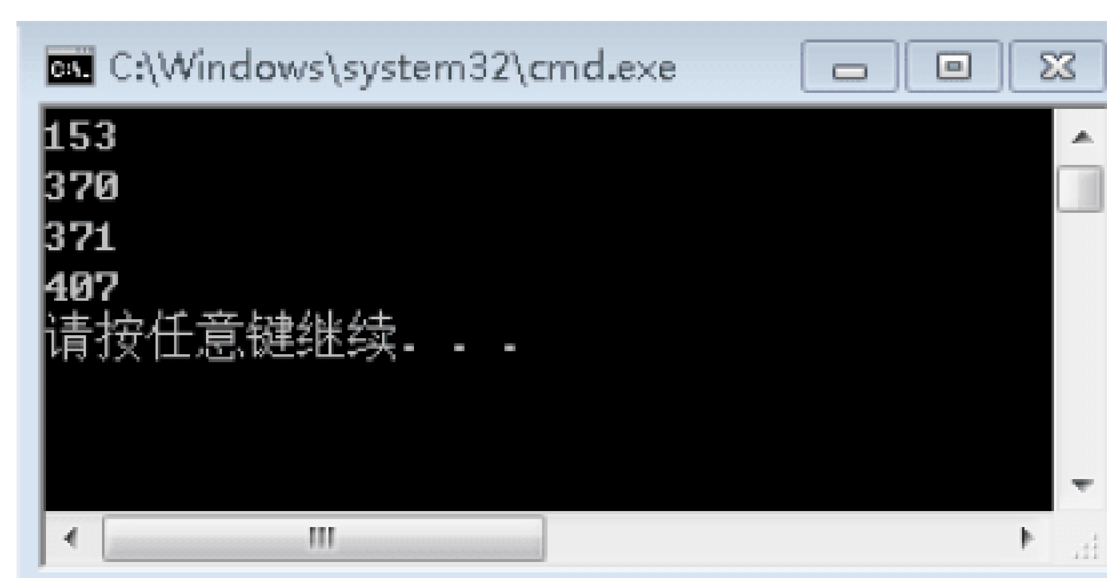


图 6.18　阿姆斯壮数

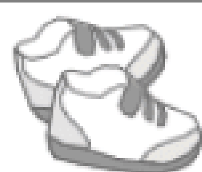
6.7.2　巴斯卡三角形

【例 6.12】 实现巴斯卡三角形。

巴斯卡三角形是两个边全输出 1，三角形的内部用上一行相邻两个数之和表示，代码如下：

实例位置：光盘\MR\Instance\06\6.12

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
using namespace std;
long combi(int n,int r)
{
	int i;
	long p = 1;
	for(i = 1;i<=r;i++)
		p = p*(n-i+1)/i;
	return p;
}
void main()
{
	int n,r,t;
	for(n = 0;n<=12;n++)                            //控制行数
```





Note

```
        {
            for(r = 0;r<=n;r++)
            {
                int i;
                if(r == 0)
                {
                    for(i = 0;i<=(12-n);i++)
                        cout<<"   ";                    //每行第一个元素的位置
                }else
                    cout<<"   ";                        //每个数之间空两个格
            cout<<setw(3)<<combi(n,r);
            }
            cout<<endl;
        }
}
```

程序使用了循环嵌套来控制显示格式，变量 n 代表行数，变量 r 代表每行元素数，自定义函数 combi 计算每个位置应该放置的数。程序运行结果如图 6.19 所示。

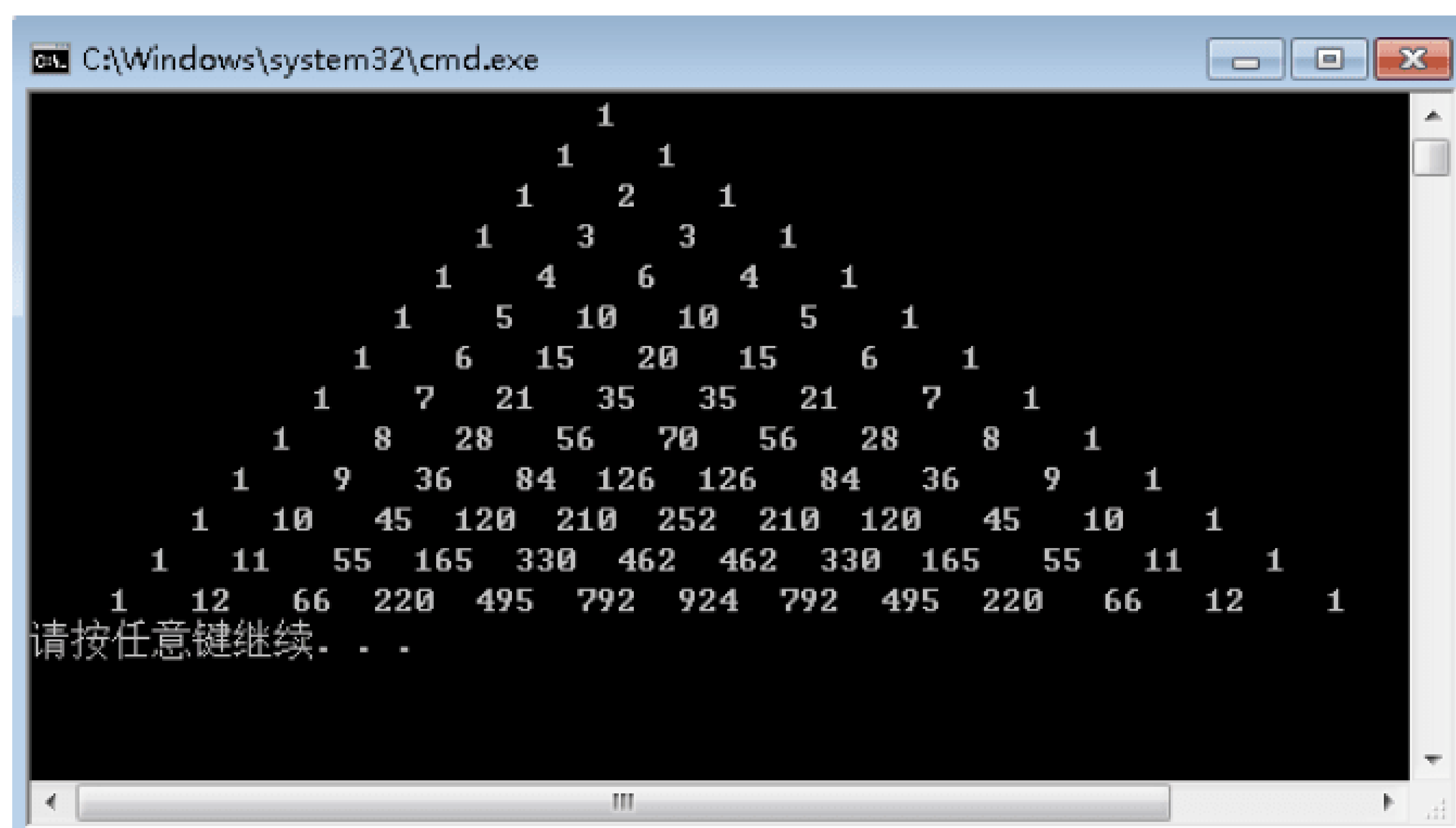


图 6.19　巴斯卡三角形

6.7.3　输出闰年

【例 6.13】 计算从 1773 年到 2012 年之间（含 1773 和 2012 年）一共有多少个闰年。设计程序输出这些闰年。本题可以对 1773 年到 2012 年之间的所有年份逐年判断闰年。也可以选择从 1773 年到 2012 年的第一个闰年开始，每隔 4 年判断一次。代码如下：

实例位置：光盘\MR\Instance\06\6.13

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main(int argc, _TCHAR* argv[])
{
```





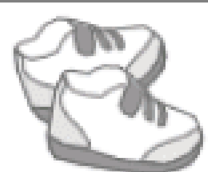
Note

```
    //若直接使用 for 循环遍历 1773-2012 年，则需要执行 240 次判断
    int year;                                   //1773 开始的第一个闰年
    int yearStart = 1773;                       //代表从何年开始
    int yearTo = 2012;                          //代表从何年结束
    //其实可以将以下 for 循环条件设定为 i<4，不过有些年份在世纪末，设定为 i<8 则是考虑到了这一点
    for(int i = 0;i<8;i++ )
    {
        if( (yearStart+i)%4==0 && (yearStart+i)%100!=0 || (yearStart+i)%400==0)
        {
            year = yearStart+i;                 //此时 year 为 1773 开始的第一个闰年
            break;
        }
    }
    int count = 1;                              //闰年个数
    for(int yearIter =year;yearIter<yearTo;count++)
    {
        if(yearIter%100 == 0&&yearIter%400 != 0)
        {
            yearIter+=4;
            count--;
            continue;
        }
        cout<<yearIter<<" ";
        if(count%10 == 0)
        {
            cout<<endl;                         //每 10 个年份换行
        }
        yearIter+=4;
    }
    cout<<endl;
                                                //整个程序执行了共 62 次循环
    return 0;
}
```

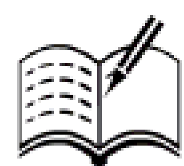
程序运行结果如图 6.20 所示。

```
D:\Windows\system32\cmd.exe
1776 1780 1784 1788 1792 1796 1804 1808 1812 1816
1820 1824 1828 1832 1836 1840 1844 1848 1852 1856
1860 1864 1868 1872 1876 1880 1884 1888 1892 1896
1904 1908 1912 1916 1920 1924 1928 1932 1936 1940
1944 1948 1952 1956 1960 1964 1968 1972 1976 1980
1984 1988 1992 1996 2000 2004 2008
请按任意键继续. . .
```

图 6.20　输出闰年



6.8 本章常见错误



Note

6.8.1 break 和 continue 语句的区别

break 是结束整个循环体，continue 是结束单次循环。例如：

```
while(x++<10)
{
	if(x==3)
	{
		break;
	}
	cout<<x;
}
```

结果是输出 1 2 就退出了整个 while 循环。

但是如果使用 continue：

```
while(x++<10)
{
	if(x == 3)
	{
		continue;
	}
	cout<<x;
}
```

结果是：1 2 4 5 6 7 8 9 10

可见这个例子不仅仅是不输出 3，便结束了本次循环，还向下执行。

6.8.2 goto 的问题

goto 当然最好不用，但在跳出深度循环方面还是可以简化问题的，只要没有因此跳出变量初始化之类的，应该没问题。批判 goto，可以先存异。

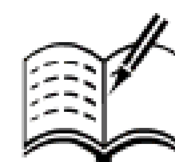
6.9 本 章 小 结

本章主要介绍了 for、while、do…while 共 3 种循环，其中使用比较灵活的是 for 循环，比较简单的是 while 循环。同样一个目标使用 3 种循环都可以实现，最终选择哪种循环来实现要根据



每个开发人员对需求的理解，但一般建议使用 for 循环。

6.10　跟 我 上 机



Note

参考答案：光盘\MR\跟我上机

对输入的分数进行排名。

通过输入不同的分数，然后计算不同分数之间的排名，实现代码如下：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int score[101] = {0};
	int juni[102] = {0};
	int count = 0,i;
	do{
		cout<<"input score:";
		cin>>score[count++];
	}while(score[count-1]!=-1);
	count--;
	for(i = 0;i<count;i++)
		juni[score[i]]++;
	juni[101] = 1;
	for(i = 100;i>=0;i--)
		juni[i] = juni[i]+juni[i+1];
	cout<<"Reuslt:"<<endl;
	for(i = 0;i<count;i++)
	{
		cout<<score[i]<<"is";
		cout<<juni[score[i]+1]<<endl;
	}
}
```

第7章

封装函数使程序模块化

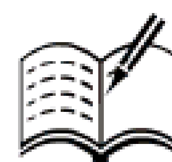
（视频讲解：1小时）

程序是由函数组成的，一个函数就是程序中的一个模块。函数可以相互调用，可以将相互联系密切的语句都放到一个函数内，也可以将复杂的函数分解成多个子函数。函数本身也有很多特点，熟练掌握函数的特点可以将程序的结构设计得更合理。

本章能够完成的主要范例（已掌握的在方框中打勾）

- ☐ 声明、定义函数
- ☐ 调用默认参数的函数
- ☐ 使用函数的递归调用解决汉诺塔问题
- ☐ 利用循环求n的阶乘
- ☐ 使用重载函数
- ☐ 输出不同生命周期的变量值
- ☐ 使用static变量实现累加





7.1　函数概述

函数就是能够实现特定功能的程序模块，它可以是只有一条语句的简单函数，也可以是包含许多子函数的复杂函数；有别人写好的存放在库里的库函数，也有开发人员自己写的自定义函数；函数根据功能可以分为字符函数、日期函数、数学函数、图形函数、内存函数等。一个程序可以只有一个主函数，但不可以没有函数。

7.1.1　定义函数

函数是有具体用途的代码块，由函数名、函数体、返回值、类型标识符以及参数列表构成。函数定义的形式如下：

```
类型标识符  函数名(参数列表)
{
        变量的声明                    //函数体内部
        语句                          //函数体内部
}
```

函数名的命名规则与变量相同。函数体是执行语句的具体内容。类型标识符是返回值的类型，返回值是函数通过 return 语句，向调用它的主调函数返回的值。返回值的类型一定要与类型标识符相对应，否则程序将不能执行。在声明或者定义一个函数时，参数列表被称作形参列表，在使用时称为实参列表。实参和形参可以看作是传递的关系，实参从函数外部传递进函数内部转变成形参，函数会使用形参的数据来执行函数体内部的语句。

7.1.2　声明和使用函数

在程序中经常看到的 main 函数就是一种函数声明，例如：

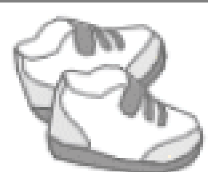
```
int main( )                          //函数名 main 与标识符 int，形参列表为空
{
        Int a= 3;
        Int b= 4;
        return 0;                    //返回值为 0，与 int 相对应
}
```

下面通过实例来介绍如何在程序中声明、定义和使用函数。

【例 7.1】 声明、定义和使用函数。

实例位置：光盘\MR\Instance\07\7.1

```
#include <iostream>
using namespace std;
```





Note

```
void ShowMessage();                 //函数声明语句
void ShowAge();                     //函数声明语句
void ShowIndex();                   //函数声明语句
void main()
{
	ShowMessage();                  //函数调用语句
	ShowAge();                      //函数调用语句
	ShowIndex();                    //函数调用语句
}
void ShowMessage()
{
	cout << "HelloWorld!" << endl;
}
void ShowAge()
{
	int iAge=23;
	cout << "age is :" << iAge << endl;
}
void ShowIndex()
{
	int iIndex=10;
	cout << "Index is :" << iIndex << endl;
}
```

程序运行结果如图 7.1 所示。



图 7.1　运行结果

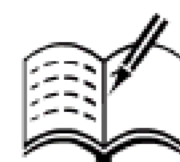
程序定义和声明了 ShowMessage、ShowAge、ShowIndex，并进行了调用，通过函数中的输出语句进行输出。

7.2　函数的参数

7.2.1　形参与实参

函数定义时如果参数列表为空，说明函数是无参函数；如果参数列表不为空，就称为带参数函数，带参数函数中的参数在函数声明和定义时被称为“形式参数”，简称形参，在函数被调用





时被赋予具体值，具体的值被称为“实际参数”，简称实参。形参与实参如图 7.2 所示。

实参与形参的个数应相等，类型应一致。实参与形参按顺序对应，函数被调用时会一一传递数据。形参与实参的区别如下：

（1）在定义函数中指定的形参，在未出现函数调用时，它们并不占用内存中的存储单元。只有在发生函数调用时，函数的形参才被分配内存单元，在调用结束后，形参所占的内存单元也被释放。

（2）实参应该是确定的值。在调用时将实参的值赋值给形参，如果形参是指针类型，就将地址值传递给形参。

```
                 形参
int function(int a,int b);
void main()
{                实参
    function(3,4);
    cout<<"Hello Word!!"<<endl;
}
int function(int a, int b)
{
    return a+b;
}
```

图 7.2　形参与实参

（3）实参与形参的类型应相同。

（4）实参与形参之间是单项传递，只能由实参传递给形参，而不能由形参传回来给实参。

实参与形参之间存在一个分配空间和参数值传递的过程，这个过程是在函数调用时发生的，C++支持引用型变量，引用型变量则没有值传递的过程，这将在后文讲到。

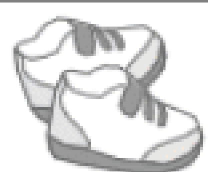
7.2.2　设置默认值

在调用有参函数时，如果经常需要传递同一个值到调用函数，在定义函数时，可以为参数设置一个默认值，这样在调用函数时可以省略一些参数，此时程序将采用默认值作为函数的实际参数。下面的代码定义了一个具有默认值参数的函数。

【例 7.2】 调用默认参数的函数。

实例位置：光盘\MR\Instance\07\7.2

```
#include "stdafx.h"
#include <iostream>
using std::cout;
using std::endl;
bool Less(int a,int b = 1)                //b 具有默认值 1
{
        if(a>b)
            return true;
        else
            return false;
}
int main()
{
int k =3;
        bool p;
        p=Less(k);                        //调用函数，只传递一个参数，第二个参数默认
        if(p)
```





Note

```
        {
                cout<<"k 大于默认参数"<<endl;
        }
        else
        {
                cout<<"k 小于默认参数"<<endl;
        }
        p = Less(k,4);
        if(p)
        {
                cout<<"k 大于参数 b"<<endl;
        }
        else
        {
                cout<<"k 小于参数 b"<<endl;
        }
                return 0;
}
```

程序运行结果如图 7.3 所示。



图 7.3　运行结果

7.3　从函数中返回

7.3.1　函数返回值

函数的返回值是指函数被调用之后，执行函数体中的程序段所取得的并返回给主调函数的值，函数的返回值通过 return 语句返回给主调函数。return 语句的一般形式如下：

```
return(表达式);
```

语句将表达式的值返回给主调函数。

关于返回值的说明：

（1）函数返回值的类型和函数定义中函数的类型标识符应保持一致。如果两者不一致，则以函数类型为准，自动进行类型转换。

（2）如函数值为整型，在函数定义时可以省去类型标识符。

（3）在函数中允许有多个 return 语句，但每次调用只能有一个 return 语句被执行，因此只





能返回一个函数值。

（4）不返回函数值的函数，可以明确定义为“空类型”，类型标识符为 void。例如：

```
void ShowIndex()
{
        int iIndex=10;
        cout << "Index is :" << iIndex << endl;
}
```

（5）类型标识符为 void 的函数不能进行赋值运算及值传递。例如：

```
i= ShowIndex();                  //不能进行赋值
SetIndex(ShowIndex);             //不能进行值传递
```



注意：

为了降低程序出错的几率，凡不要求返回值的函数都应定义为空类型。

7.3.2　了解空函数

没有参数和返回值、函数的作用域为空的函数就是空函数。

```
void setWorkSpace(){ }
```

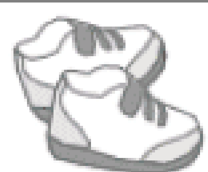
调用此函数时，什么工作也不做，没有任何实际意义。在主函数 main 中调用 setWorkSpace 函数时，这个函数没有起到任何作用。例如：

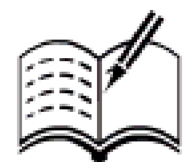
```
void setWorkSpace(){ }
void main()
{
     setWorkSpace();
}
```

空函数存在的意义是：在程序设计中往往需要根据情况设计若干模块，分别由一些函数来实现。而在第一阶段只设计最基本的模块，其他一些次要功能或锦上添花的功能则在以后需要时陆续补上。在编写程序的开始阶段，可以在将来准备扩充功能的地方写上一个空函数，这些函数没有开发完成，先占一个位置，以后用一个编好的函数代替它。这样做，程序的结构清楚，可读性好，以后扩充新功能方便，对程序结构影响不大。

7.4　递归调用函数

直接或间接调用自己的函数被称为递归函数（recursive funciton）。





使用递归方法解决问题的特点是：问题描述清楚、代码可读性强、结构清晰，代码量比使用非递归方法少。缺点是递归程序的运行效率比较低，无论是从时间角度还是从空间角度都比非递归程序差。对于时间复杂度和空间复杂度要求较高的程序，使用递归函数调用要慎重。

递归函数必须定义一个停止条件，否则函数永远递归下去。

有 3 个立柱垂直矗立在地面，给这 3 个立柱分别命名为 A、B、C。开始时立柱 A 上有 64 个圆盘，这 64 个圆盘大小不一，并且按从小到大的顺序依次摆放在立柱 A 上。现在的问题是要将立柱 A 上的 64 个圆盘移到立柱 C 上，并且每次只允许移动一个圆盘，在移动过程中始终保持大盘在下，小盘在上。

分析程序：

假设移动 4 个圆盘，立柱 A 上的圆盘按由上到下的顺序分别命名为 a、b、c、d，如图 7.4 所示。

先考虑将 a 和 b 移动到立柱 C 上。移动顺序是 a→B，b→C，a→C，移动结果如图 7.5 所示。

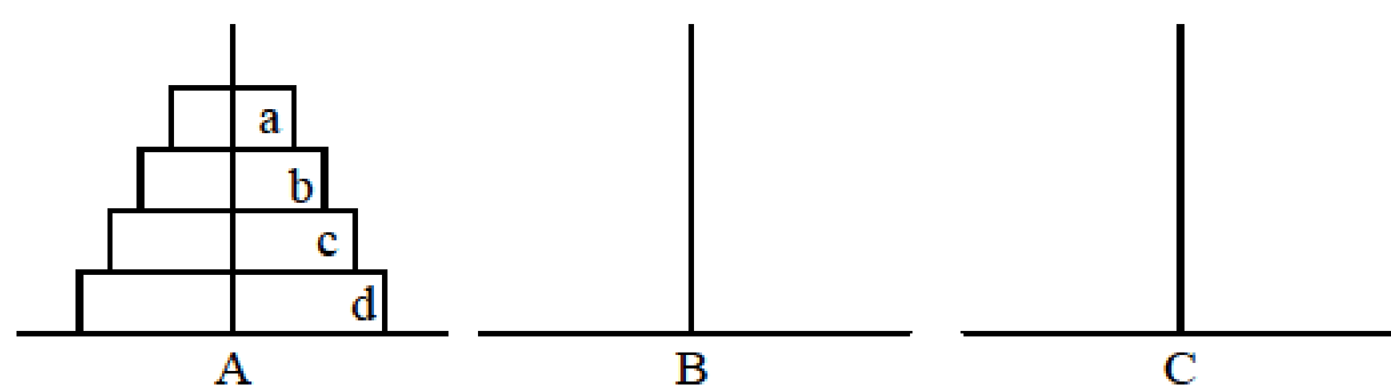


图 7.4　圆盘原始状态

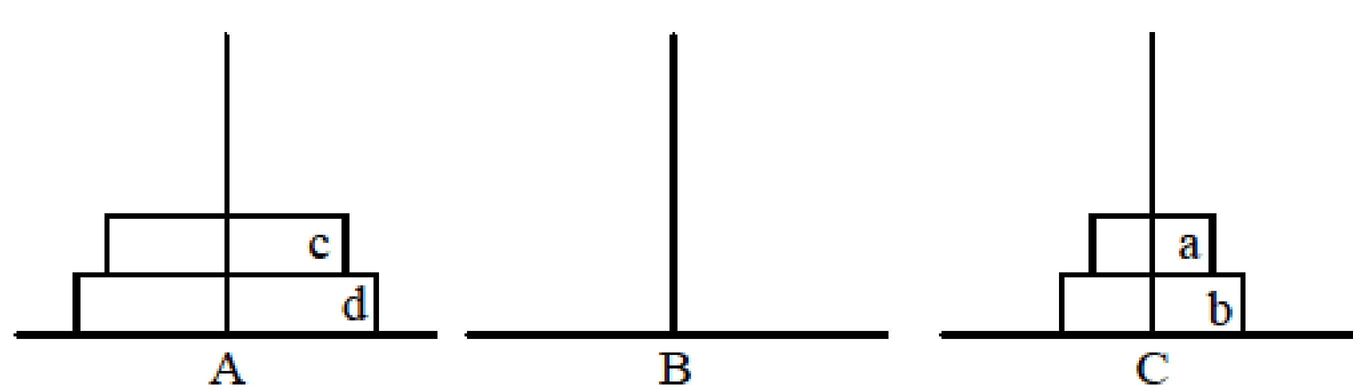


图 7.5　移动两个圆盘到目标

如果要将 c 也移动到 C 上，就要暂时将 c 移动到 B，然后再移动 a 和 b。移动顺序是 c→B，a→A，b→B，a→B，d→c，移动结果如图 7.6 所示。

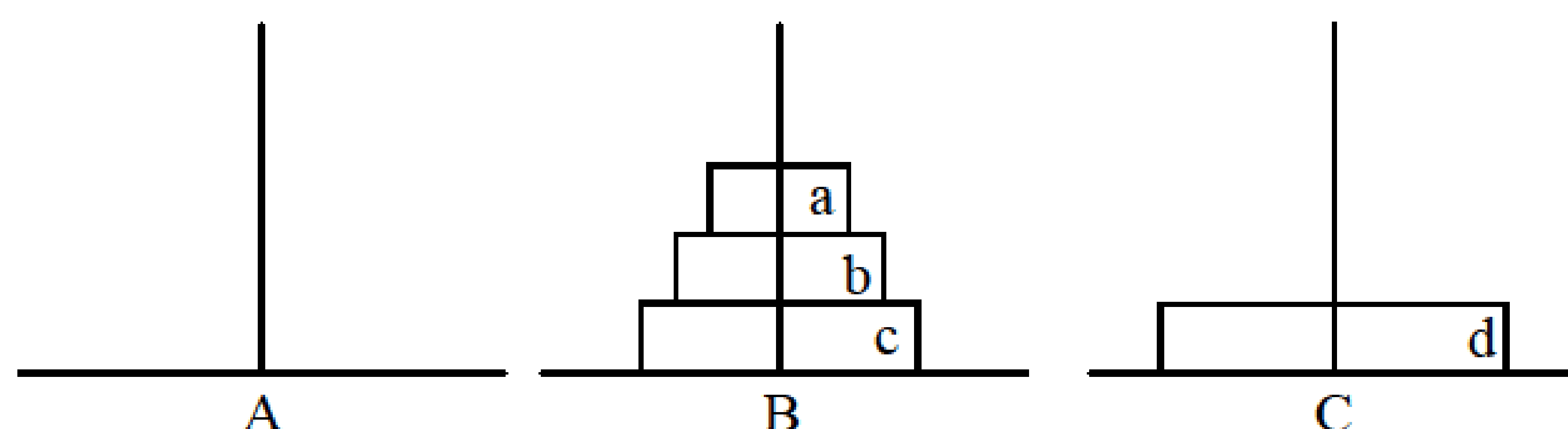


图 7.6　移动 3 个圆盘到目标

最后是完成 4 个圆盘的移动，移动顺序是 a→C，b→A，a→A，c→C，a→B，b→C，a→C。

总结一下，要将 4 个圆盘移动到指定立柱总共需要移动 15 次。在移动过程中将两个圆盘移动到指定立柱需要移动 3 次，分别是 a→B，b→C，a→C。在移动过程中将 3 个圆盘移动到指定立柱需要移动 7 次，分别是 a→B，b→C，a→C，c→B，a→A，b→B，a→B。移动次数可以总结为是 2^n-1 次。在移动过程中可以将 a、b、c 这 3 个圆盘看成是一个圆盘，移动 4 个圆盘的过程就像是在移动两个圆盘。还可以将 a、b、c 这 3 个圆盘中的 a、b 两个圆盘看成是一个圆盘，移动 3 个圆盘也像是在移动两个圆盘。可以使用递归的思路来移动 n 个圆盘。

移动 n 个圆盘可以分成 3 个步骤：

（1）把 A 上的 n-1 个圆盘移到 B 上。

（2）把 A 上的一个圆盘移到 C 上。

（3）把 B 上的 n-1 个圆盘移到 C 上。

【例 7.3】 汉诺（Hanoi）塔问题。





实例位置：光盘\MR\Instance\07\7.3

```
#include "stdafx.h"
#include <iostream>
using namespace std;
long lCount;
void move(int n,char x,char y,char z)                  //将 n 个圆盘从 x 针借助 y 针移到 z 针上
{
	if(n==1)
		cout << "Times:" << ++lCount << x << "->" << z << endl;
	else
	{
		move(n-1,x,z,y);                        //递归调用
		cout << "Times:" << ++lCount << x << "->" << z <<endl;
		move(n-1,y,x,z);
	}
}
void main()
{
	int n ;
	lCount=0;
	cout << "please input a number" << endl;
	cin >> n ;
	move(n,'a','b','c');                            //调用 move 函数
}
```

程序运行结果如图 7.7 所示。

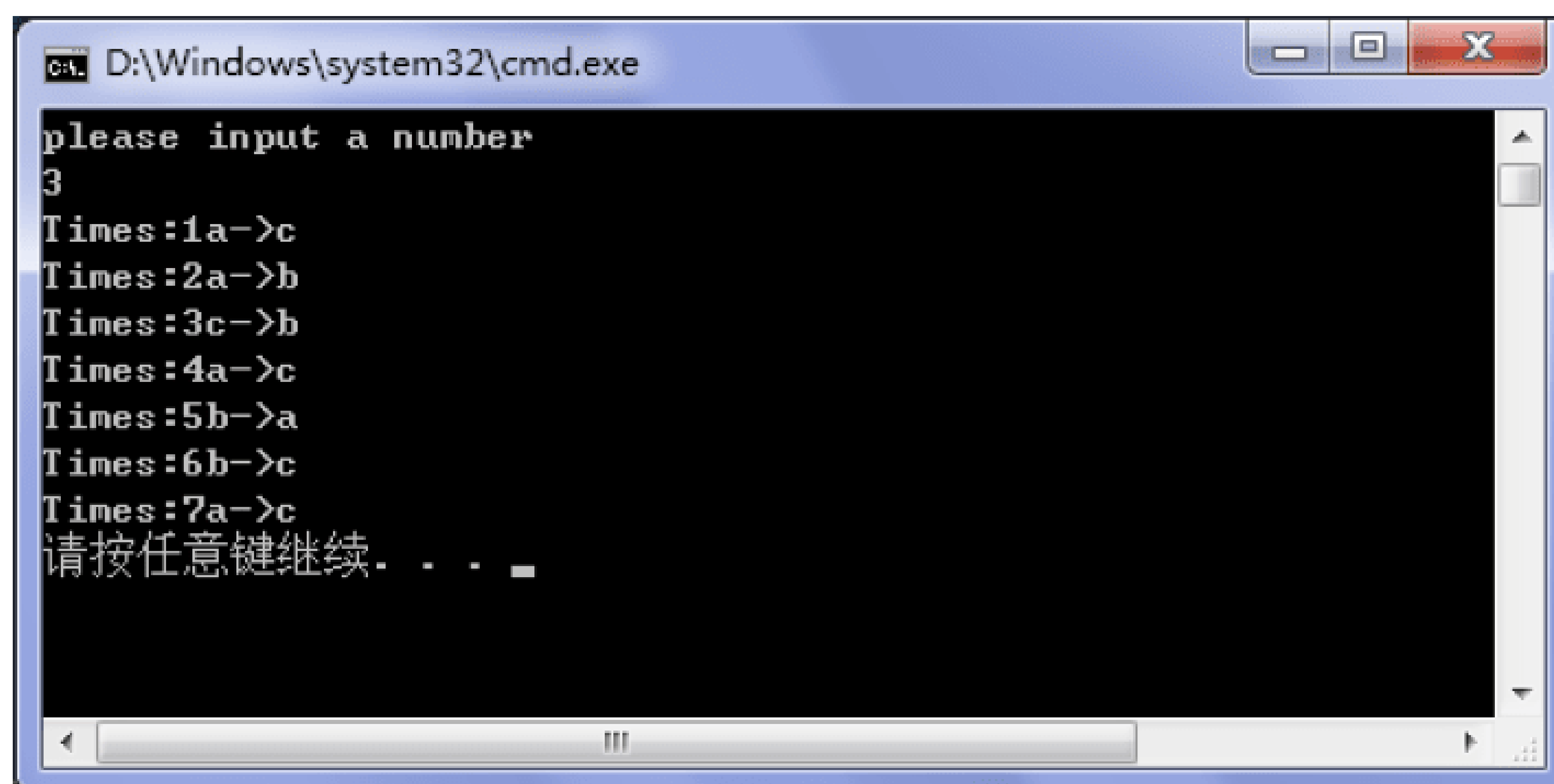


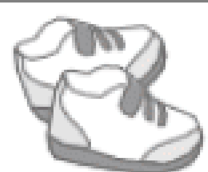
图 7.7　运行结果

输入数字 3，表示移动 3 个圆盘，程序打印出挪动 3 个圆盘的步骤。

【例 7.4】　求 n 的阶乘。

实例位置：光盘\MR\Instance\07\7.4

```
#include "stdafx.h"
#include <iostream>
using namespace std;
long Fac(int n)
```





Note

```
{
        if(n==0)
                return 1;
        else
                return n*Fac(n-1);                    //递归调用
}
void main()
{
        int n ;
        long f;
        cout << "please input a number" << endl;
        cin >> n ;
        f=Fac(n);                                     //调用 Fac 函数
        cout << "Result :" << f << endl;
}
```

程序运行结果如图 7.8 所示。

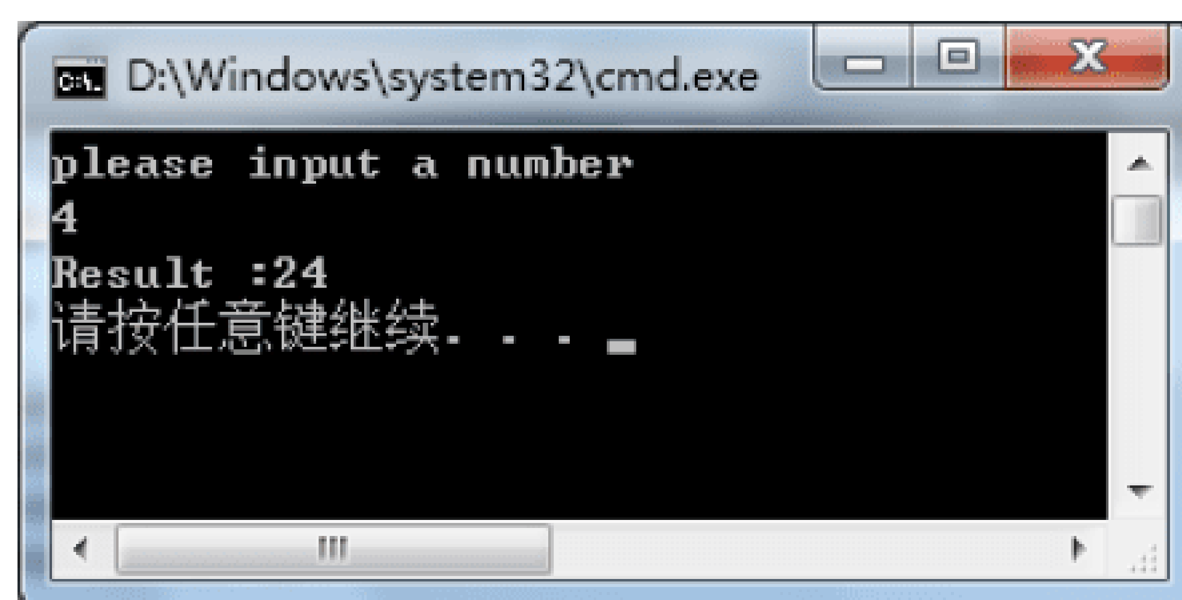


图 7.8　计算阶乘

程序中 Fac 函数实现了计算 n 的阶乘。以 n 等于 4 为例，4!等于 4*3!，3!等于 3*2!，……，1!等于 1。当计算 4 的阶乘时，只要知道 3 的阶乘就可以了，4*3!等于 4!。同理，计算 3 的阶乘，只要知道 2 的阶乘就可以了，依此类推。1 的阶乘为 1，知道了 1 的阶乘，就可以计算 2 的阶乘，知道 2 的阶乘就可以计算 3 的阶乘……

在上面的递归函数中，如果传递一个很大的数作为参数，会导致堆栈溢出，因为每调用一个函数，系统会为函数的参数分配堆栈空间。对于上述的递归函数 Fac，完全可以用连续乘积的方式实现。

【例 7.5】 利用循环求 n 的阶乘。

实例位置：光盘\MR\Instance\07\7.5

```
#include "stdafx.h"
#include <iostream>
using namespace std;
typedef unsigned int UINT;                    //自定义类型
long Fac(const UINT n)                        //定义函数
{
        long ret = 1;                         //定义结果变量
        for(int i=1; i<=n; i++)               //累计乘积
        {
                ret *= i;
        }
        return ret;                           //返回结果
}
void main()
{
        int n ;
```





```
        long f;
        cout << "please input a number" << endl;
            cin >> n ;
        f = Fac(n);
        cout << "Result :" << f << endl;
}
```

程序运行结果如图 7.9 所示。

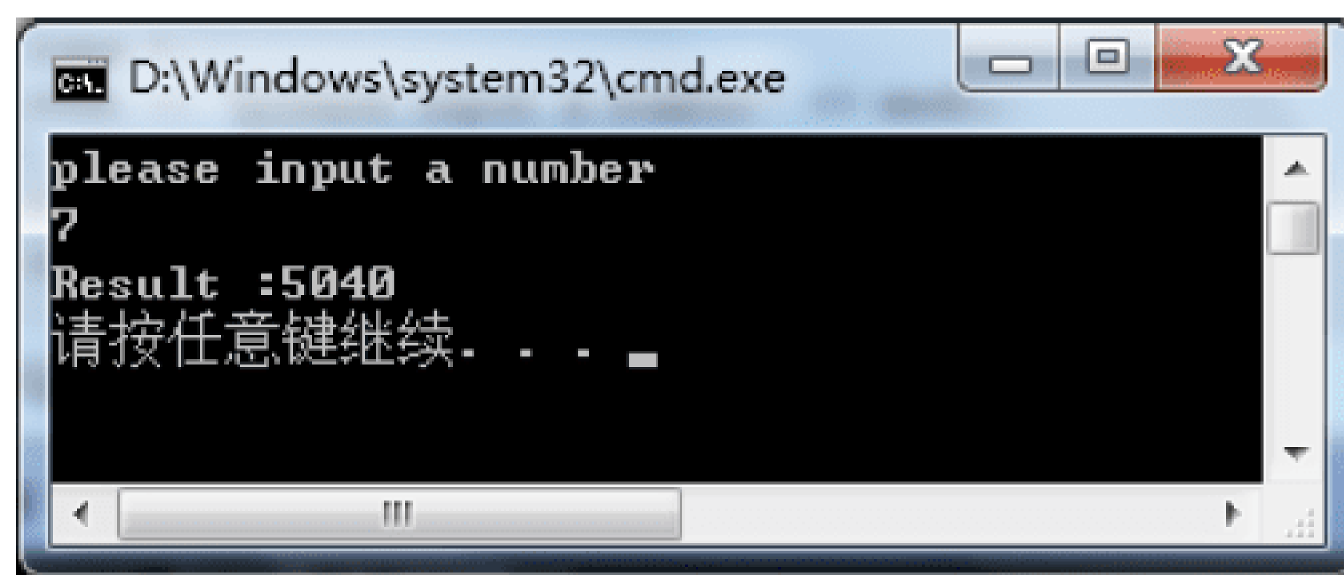
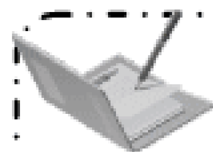


图 7.9　利用循环求 n 的阶乘



说明：

在编程算法中，实例 7.4 计算阶乘的方法称为递归法，实例 7.5 的方法称为迭代法。对这些算法有兴趣深入研究的读者可以查阅相关资料。

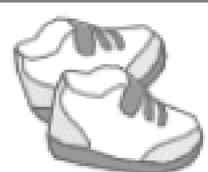
7.5　重载函数的使用

C++中使用了名字重组的技术，通过函数的参数类型来识别函数，所谓重载函数就是指多个函数具有相同的函数标识名，而参数类型或参数个数不同。函数调用时，编译器以参数的类型及个数来区分调用哪个函数。下面实例定义了重载函数。

【例 7.6】 使用重载函数。

实例位置：光盘\MR\Instance\07\7.6

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int Add(int x ,int y)                       //定义第一个重载函数
{
        cout << "int add" << endl;          //输出信息
        return x + y;                       //设置函数返回值
}
double Add(double x,double y)               //定义第二个重载函数
{
        cout << "double add" << endl;       //输出信息
        return x + y;                       //设置函数返回值
}
```



```
int main()
{
	int ivar = Add(5,2);			//调用第一个 Add 函数
	float fvar = Add(10.5,11.4);		//调用第二个 Add 函数
	return 0;
}
```



Note

程序运行结果如图 7.10 所示。



图 7.10　函数重载

程序中定义了两个相同函数名标识符的函数，函数名都为 Add，在 main 调用 Add 函数时实参类型不同，语句“int ivar = Add(5,2);”的实参类型是整型，语句“float fvar = Add(10.5,11.4);”的实参类型是浮点型，编译器可以区分这两个函数，并正确调用相应的函数。

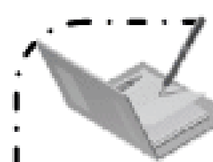


技巧：

可以看出，函数重载不仅是用来区分重名函数的，更重要的是它为我们解决了要使用同名函数解决不同参数类型返回值类型的问题。同样是求和函数，我们还可以定义一个浮点数类型的函数，这样求和函数 Add 在代码中的重用性就会变高，思考起来也更加方便。

在定义重载函数时，应注意函数的返回值类型不作为区分重载函数的一部分。下面的函数重载是非法的：

```
int Add(int x ,int y)			//定义一个重载函数
{
	return x + y;
}
double Add(int x,int y)			//定义一个重载函数
{
	return x + y;
}
```



说明：

C++中的重载、重写和覆盖的区别如下。

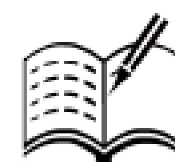
☑ 重载：同一个作用域（参数的类型或者个数不同）和一个类中。

☑ 覆盖：不同作用域，发生在父类和子类之间。

☑ 重写：前面有 virtual（虚函数可以被重写，在继承中，派生类）。



7.6　生存周期与作用域



Note

变量的声明位置以及储存方式有很多类别，这些都影响着函数对变量的调用。

7.6.1　变量的作用域

依据变量的声明位置可以将变量分为局部变量以及全局变量。在函数体内部声明的变量，称为局部变量。在所有函数体外部定义的变量称为全局变量。

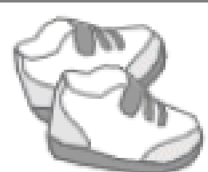
【例 7.7】 变量的作用域。

实例位置：光盘\MR\Instance\07\7.7

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int globalCount = 33;               //全局变量
int GetCount();                     //声明函数
void SetCount(int k);
void main()
{
    Int count = 100;                //局部变量
    cout << globalCount<< endl;     //输出全局变量
    SetCount(200);//
    cout << GetCount() << endl;
}
void SetCount(int k)                //定义函数
{
    int hisCount;                   //定义局部变量
    //myCount =200;                 //执行会出错，注释掉
    //count =200;                   //执行会出错，注释掉
    hisCount = k;                   //函数体自身内部定义的变量可以被使用，k 也可以看作是局部变量
    globalCount=hisCount;           //给全局变量赋值

}
int GetCount()
{
    int myCount;                    //定义局部变量
    myCount = globalCount;          //使用自身的局部变量
    return myCount;
}
```

程序运行结果如图 7.11 所示。





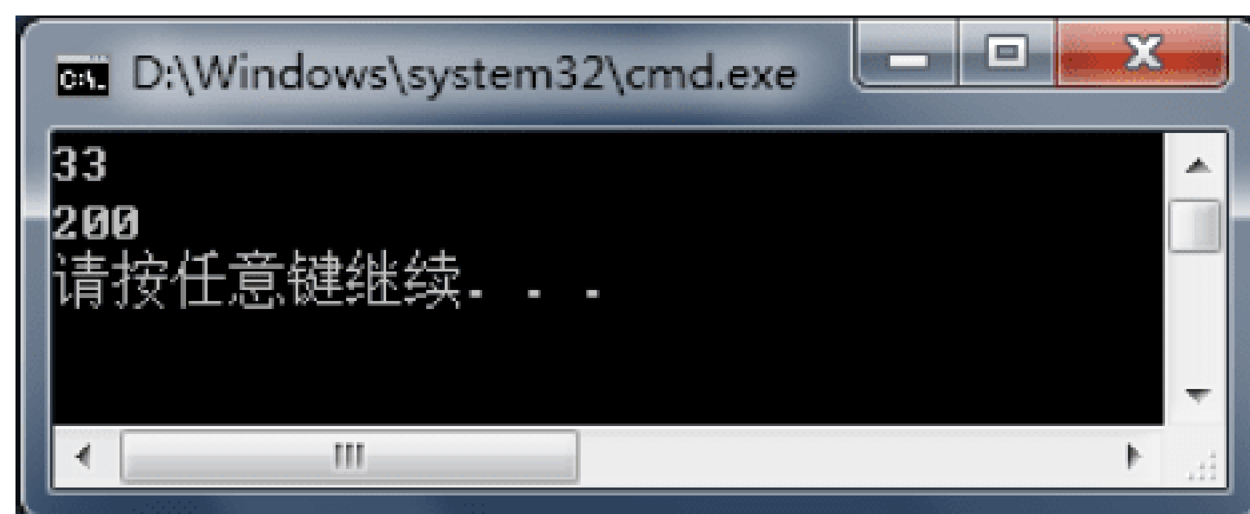


图 7.11　执行结果

当一个函数体内定义的局部变量和全局变量同名时，程序会优先选择使用局部变量。若想使用全局变量则需要在变量名前加入区域符号“::”。

【例 7.8】 同名的全局变量与局部变量。

实例位置：光盘\MR\Instance\07\7.8

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int name = 0;
int main()
{
	int name = 3;
	cout<<"局部变量 name 的值: "<<endl;
	cout<<name<<endl;
	cout<<"全局变量 name 的值: "<<endl;
	cout<<::name<<endl;
	return 0;
}
```

程序运行结果如图 7.12 所示。



图 7.12　执行结果

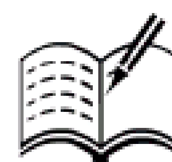
变量的作用域说明如下。

- ☑ 全局变量：从定义该变量处开始，到本文件结束。
- ☑ 静态全局变量：
  - ➢ 其他源文件可以使用相同变量名，彼此独立。
  - ➢ 在某源文件中定义的静态全局变量不能被其他源文件使用和修改。
  - ➢ 静态全局变量只能在本文件内使用，具有内部链接静态，不允许在其他文件中调用。
- ☑ 局部变量：只能由其被定义的模块内部的语句所访问，局部变量在自己的代码模块外部是无效的。
- ☑ 静态局部变量：在第一次调用时初始化，以后再进入函数时保持上一次的原值，记忆性。

7.6.2　变量的生存周期

定义在同一个函数中的变量生存周期并不完全相同。在不同语句块定义的变量，作用域的大





小也不一样。下面是各变量的生命周期。

- ☑ 全局变量：程序运行的整个周期都存在。
- ☑ 局部变量：在函数体内部或复合语句内部，函数执行完释放内存空间，从定义该变量处开始生效，模块结束，变量消亡。
- ☑ 静态局部变量：整个运行周期都存在。
- ☑ 静态全局变量：整个运行周期都存在。

7.6.3　变量的储存方式

存储类别是变量的属性之一，C++语言中定义了 4 种变量的存储类别，分别是 auto 变量、static 变量、register 变量和 extern 变量。变量存储方式不同会使变量的生存期不同，生存期表示了变量存在的时间。生存期和变量作用域是从时间和空间这两个不同的角度来描述变量的特性。

静态存储变量通常是在变量定义时就分配固定的存储单元并一直保持不变，直至整个程序结束。前面讲过的全局变量即属于此类存储方式，它们存放在静态存储区中。动态存储变量是在程序执行过程中使用它时才分配存储单元，使用完毕立即将该存储单元释放。前面讲过的函数的形式参数，在函数定义时并不给形参分配存储单元，只是在函数被调用时才予以分配，调用函数完毕立即释放，此类变量存放在动态存储区中。从以上分析可知，静态存储变量是一直存在的，而动态存储变量则时而存在时而消失。

1. 自动变量

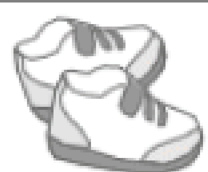
这种存储类型是 C++语言程序中默认的存储类型。函数内未加存储类型说明的变量均视为自动变量，例如：

```
{
int i,j,k;
…
}
```

自动变量具有以下特点：

（1）自动变量的作用域仅限于定义该变量的个体内。在函数中定义的自动变量，只在该函数内有效。在复合语句中定义的自动变量只在该复合语句中有效。例如：

```
int Show()
{
	int x,y;
	if(true)
	{
		char ch;
		cout << ch << endl;		//正确
		cout << x << endl;		//正确
	}
	cout << ch << endl;			//错误
```



```
    cout << x << endl;                    //正确
}
```



Note

（2）自动变量属于动态存储方式，变量分配的内存是在栈中，当函数调用结束后，自动变量的值会被释放。同样在复合语句中定义的自动变量，在退出复合语句后也不能再使用，否则将引起错误。

（3）由于自动变量的作用域和生存期都局限于定义它的个体内（函数或复合语句内），因此不同的个体中允许使用同名的变量而不会混淆。即使在函数内定义的自动变量也可与该函数内部的复合语句中定义的自动变量同名。

【例 7.9】 输出不同生命期的变量值。

实例位置：光盘\MR\Instance\07\7.9

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
{
    int i,j,k;
    cout <<"input the number:" << endl;
    cin >> i >> j;
    k=i+j;
    if( i!=0 && j!=0 )
    {
        int k;
        k=i-j;
        cout << "k :" << k << endl;        //输出变量 K 的值
    }
    cout << "k :" <<k << endl;             //输出变量 k 的值
}
```

程序运行结果如图 7.13 所示。

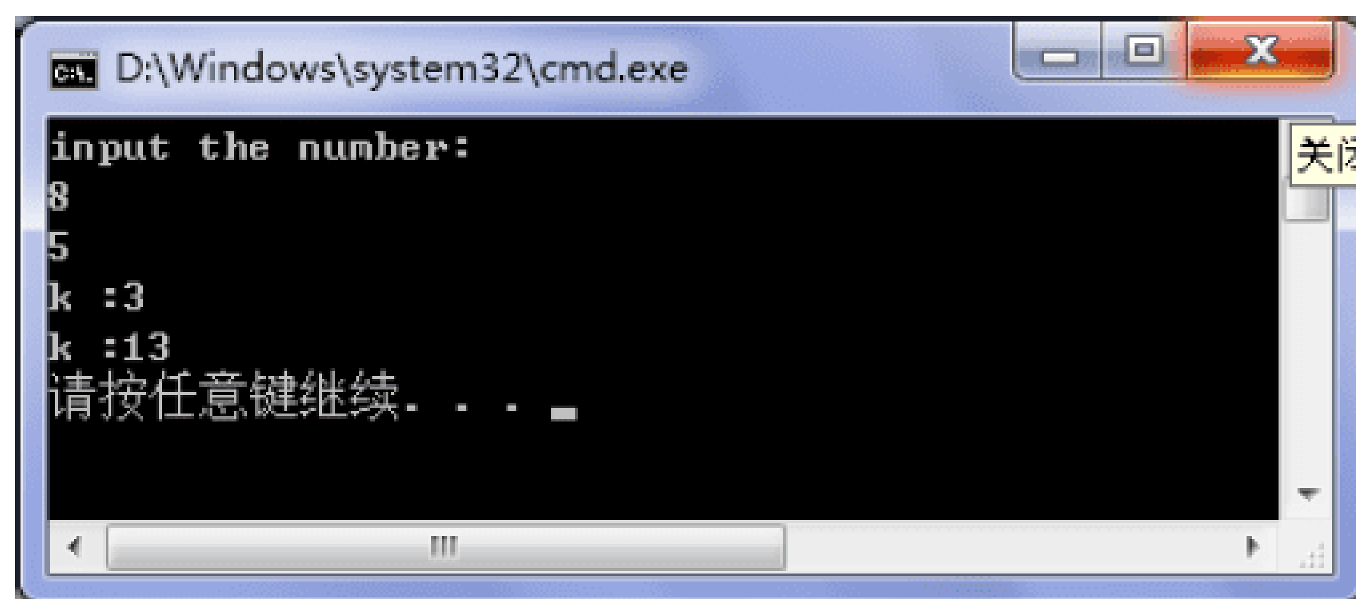


图 7.13　输出不同生命期的变量值

程序两次输出变量 k 为自动变量。第一次输出的是 i-j 的值，第二次输出的是 i+j 的值。虽然变量名都为 k，但其实是两个不同的变量。



注意：

在以前的标准中，关键字 auto 代表了自动变量的存储方式。在 C++11 的新标准中这一功能已经失效，在第 14 章中将详细介绍 auto 关键字的作用。



2．static 变量

在声明变量前加关键字 static，可以将变量声明成静态变量。静态局部变量的值在函数调用结束后不消失，静态全局变量只能在本源文件中使用。例如，声明变量为静态变量：



```
static int a,b;
static float x,y;
static int a[3]={0,1,2};
```

静态变量属于静态存储方式，它具有以下特点：

（1）静态变量在函数内定义，在程序退出时释放，在程序整个运行期间都不释放，也就是说它的生存期为整个源程序。

（2）静态变量的作用域与自动变量相同，在函数内定义就在函数内使用，尽管该变量还继续存在，但不能使用它，如再次调用定义它的函数时，它又可继续使用。

（3）编译器会为静态局部变量赋予 0 值。

下面通过实例介绍 static 变量的用法。

【例 7.10】 使用 static 变量实现累加。

实例位置：光盘\MR\Instance\07\7.10

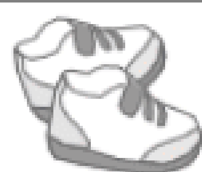
```
#include "stdafx.h"
#include<iostream>
using namespace std;
int add(int x)
{
	static int n=0;
	n=n+x;
	return n;
}
void main()
{
	int i,j,sum;
	cout << " input the number:" << endl;
	cin >> i;
	cout << "the result is:" << endl;
	for(j=1;j<=i;j++)
	{
		sum=add(j);
		cout << j << ":" <<sum << endl;
	}
}
```

程序运行结果如图 7.14 所示。

程序中 n 是静态局部变量，每次调用函数 add 时，静态局部变量 n 都保存了前次被调用后留下的值。所以当输入循环次数 3 时，变量 sum 累加的结果是 6，而不是 3。

如果去除 static 关键字，则运行结果如图 7.15 所示。

当输入循环次数为 3 时，变量 sum 累加的结果是 3。变量 n 不再使用静态存储区空间，每次





调用后变量 n 的值都被释放，再次调用时 n 的值为初始值 0。

```
D:\Windows\system32\cmd.exe
 input the number:
5
the result is:
1:1
2:3
3:6
4:10
5:15
请按任意键继续. . .
```

图 7.14　使用 static 变量实现累加

```
D:\Windows\system32\cmd.exe
 input the number:
5
the result is:
1:1
2:2
3:3
4:4
5:5
请按任意键继续. . .
```

图 7.15　运行结果

3．register 变量

通常变量的值存放在内存中，当对一个变量频繁读写时，需要反复访问内存储器，花费大量的存取时间。为了提高效率，C++语言可以将变量声明为寄存器变量，这种变量将局部变量的值存放在 CPU 的寄存器中，使用时不需要访问内存，而直接从寄存器中读写。寄存器变量的说明符是 register。

对寄存器变量的说明如下：

（1）寄存器变量属于动态存储方式。凡需要采用静态存储方式的量不能定义为寄存器变量。

（2）编译程序会自动决定哪个变量使用寄存器存储。register 起到程序优化的作用。

4．extern 变量

在一个源文件中定义的变量和函数只能被本文件中的函数调用，一个 C++程序中会有许多源文件，如果使用非本源文件的全局变量呢？C++提供了 extern 关键字来解决这个问题。在使用其他源文件的全局变量时，只需要在本源文件使用 extern 关键字来声明这个变量即可。

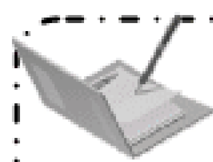
在 Sample1.cpp 源文件中定义全局变量 a、b、c，代码如下：

```
int a,b;      /*外部变量定义*/
char c;       /*外部变量定义*/
vid main()
{
      cout << a << endl;
      cout << b << endl;
      cout << c << endl;
}
```

在 Sample2.cpp 源文件中要使用 Sample1.cpp 源文件中全局变量 a、b、c，代码如下：

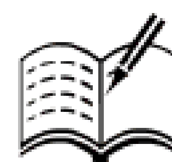
```
extern int a,b;   /*外部变量说明*/
extern char c;    /*外部变量说明*/
func (int x,y)
{
      cout << a << endl;
      cout << b << endl;
      cout << c << endl;
}
```





说明：

在第 2 章中曾简单介绍过头文件和源文件的作用，在以后的章节中，将更加深入地探讨它们的使用方法。



Note

在 Sample2.cpp 源文件中，编译系统不再为全局变量 a、b、c 分配内存空间，而是改变全局变量 a、b、c 的值，在 Sample1.cpp 源文件中输出值也会发生变化。

7.7　名称空间

命名空间，也称为名字空间，英文关键字为 namespace。在以前的代码中经常使用这样一个语句：

```
using namespace std;
```

我们要使用标准输入/输出流，除了包含它们所在的头文件之外，还必须使用它们的名称空间。实质上，namespace 后面的 std 正是该名称空间的名称。它的主要作用就是防止不同文件中包含的同一变量、函数等因名字重复而导致的错误。using namespace std 表示的是在本文件中使用所有名字为 std 空间的所有数据，而不需要像下面这样加上名称标识：

```
using std::cout;
using std::endl
```

除了上述两种方法，最常用的方法可以是如下形式：

```
std::cout<<"hi!!"<<endl;
```

将这 3 种方法比较：

☑ 第一种方法使用简便，编程者不需要逐个包含名称空间中的变量、函数等，可以直接使用它们。缺点是在文件中，失去了名称空间应有的作用，定义需注意与该名称空间中的各个数据命名冲突问题。

☑ 第二种方法比较折中，编程者为了方便地使用名称空间的少数数据而使用的方法。

☑ 第三种方法在每次使用名称空间数据时都要加上名称空间的名字，引用起来比上述两种方法稍显繁琐。但这种方法在所有情况下都适用，不会造成混乱，在编写大型项目时比较可取。

定义一个名称空间可以使用 namespace 关键字，形式如下：

```
namespace 名称空间名{
      代码
}
```

【例 7.11】 名称空间的定义和使用。



实例位置：光盘\MR\Instance\07\7.11



Note

```
#include "stdafx.h"
#include <iostream>
using std::cout;                                        //头文件包含了 std 名称空间，可以使用
using std::endl;
namespace welcome{
        int count = 3;
        float getCount(){
                return 3.33f;
        }
}
using namespace welcome;                                //定义名称空间之后使用
namespace hello{
        int count =4;
        float getCount(){
                return 4.44f;
        }
}
float getCount(){
        return 1.11f;
}
int main()
{
        int count =1;
        cout<<"直接调用 getCount 函数和使用 count"<<endl;
        //cout<<"getCount:"<<getCount()<<endl;     编译器函数重载冲突
        cout<<"count:"<<count<<endl;
        cout<<"--------------------------------------"<<endl;   //视觉分割线
        cout<<"使用域标识符调用 getCount 函数和使用 count"<<endl;
        cout<<"::getCount:"<<::getCount()<<endl;        //没有定义名称空间
        cout<<"count:"<<::count<<endl;
        cout<<"welcome::getCount:"<<welcome::getCount()<<endl;
        cout<<"welcome::count:"<<welcome::count<<endl;
        cout<<"::getCount:"<<hello::getCount()<<endl;
        cout<<"hello::count:"<<hello::count<<endl;
        return 0;
}
```

程序运行结果如图 7.16 所示。

```
D:\Windows\system32\cmd.exe
直接调用getCount函数和使用count
count:1
--------------------------------------
使用域标识符调用getCount函数和使用count
::getCount:1.11
count:3
welcome::getCount:3.33
welcome::count:3
::getCount:4.44
hello::count:4
请按任意键继续. . .
```

图 7.16　运行结果





7.8　综 合 应 用

7.8.1　等差数列求和

【例 7.12】 在等差数列中，后一个数和前一个数的差是固定的。例如，1、5、9、13…是一个差值是 4 的等差数列。本实例实现编写一个函数求出等差数列前 n 项的和。在主函数中调用 sum 函数，累加每一项的值，得出前 n 项的和并返回。关键代码如下：

实例位置：光盘\MR\Instance\07\7.12

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int Sum(int a1,int d,int count)
{
	int sum = 0;
	for(int i= 0;i<count;i++)			//这里也可以直接利用公式计算
	{
		sum += a1+i*d;			//累加每一项的和并赋给 sum
	}
	return sum;				//返回前 n 项的和
}
int main(int argc, _TCHAR* argv[])
{
	int a1,d,count,sum;
	cout<<"输入等差数列第一项"<<endl;
	cin>>a1;
	cout<<"输入等差数列差值"<<endl;
	cin>>d;
	cout<<"输入等差数列项数"<<endl;
	cin>>count;
	sum = Sum(a1,d,count);
	cout<<"这个等差数列的前"<<count<<"项的和为"<<sum<<endl;
	return 0;
}
```

程序运行结果如图 7.17 所示。

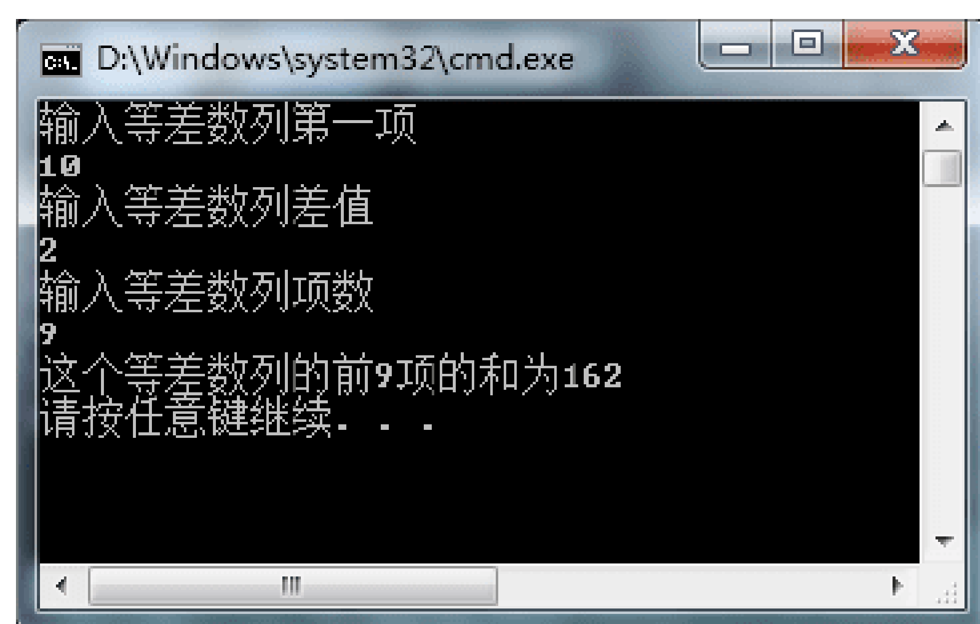
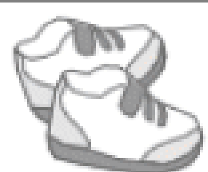


图 7.17　等差数列求和



7.8.2 提款机的记录



Note

【例 7.13】 本实例将设计一个函数 ATM 模拟取款机的提款过程，在函数体内记录了自身现金剩余量。主函数用一个循环不断地访问它，每次访问该函数都会输出提款次数，当现金小于 0 时，提示余额不足，跳出循环，停止访问。关键代码如下：

实例位置：光盘\MR\Instance\07\7.13

```
#include "stdafx.h"
bool ATM(int cash)
{
	static int myCash = 10000;		//定义静态变量作为剩余现金，变量不会随着函数结束而被释放
	myCash -= cash;			//每次提款，现金会减少
	if(myCash<0)			//判断是否余额不足
	{
		return false;		//余额不足，不能继续提取
	}
	else
	{
		return true;		//还有现金
	}
}
int main(int argc, _TCHAR* argv[])
{
	int i=0;
	while(ATM(1000))
	{
		i++;
		printf("取款%d 次\n",i);
	}
	printf("ATM 里没钱了\n");
	return 0;
}
```

程序运行结果如图 7.18 所示。

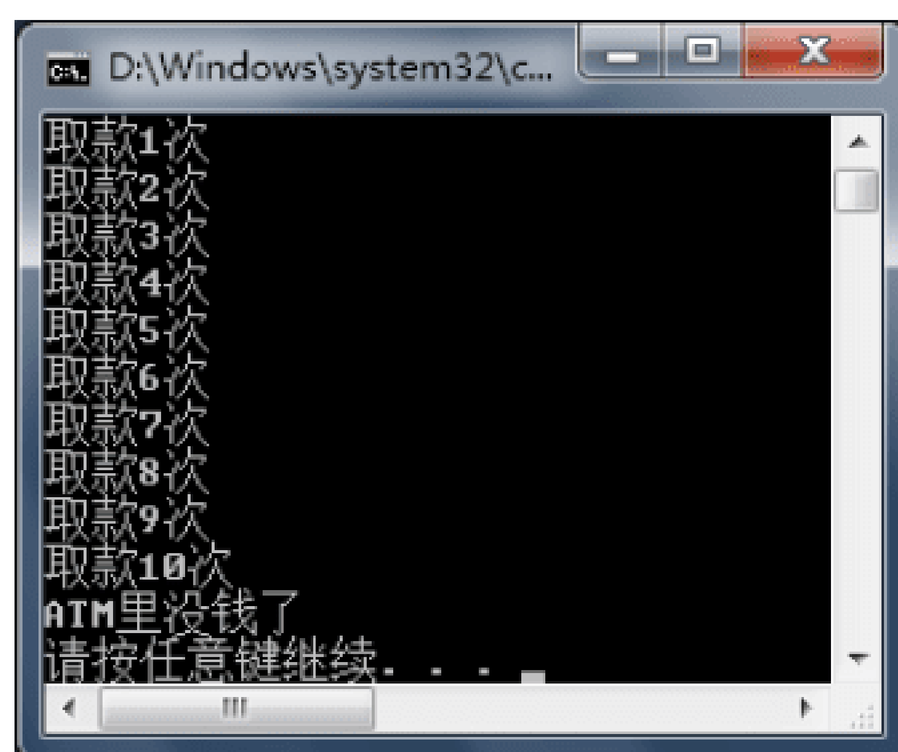


图 7.18 提款机的记录



7.9　本章常见错误



7.9.1　函数中返回的数组地址无效

在调用函数时，返回一个数组的地址，主函数却无法获取数组中的内容。例如：

```
char* fun()
{
    char str[10] = {'\0'};
    char *p = str;
    strcpy(str,"123456");              //在数组 str 中存储了字符串“123456”
    printf("in fun:%s\n",str);         //在 fun 函数中可以正常显示数组内容
    return p;                          //向主函数返回该数组的首地址
}
int main()
{
    printf("in main:%s\n",fun());      //在主函数中输出数组 str 的内容时出现段错误
    return 0;
}
```

这是因为 str 是局部数组。函数中定义的变量是局部变量，存储在内存的栈区。作用域和声明周期都是在本模块内有效，函数调用完毕，该模块结束，里面定义的变量（或数组）所占内存被释放，因此在主函数中得到的返回地址已经无效。此时再按此地址输出，将会访问不可用内存，出现段错误。

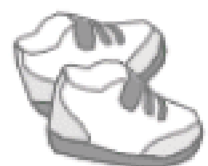
解决方法为：可将该数组定义为静态数组或全局数组。

7.9.2　声明函数时不要忘记加分号

如果一个函数在主函数后面定义，那么在调用该函数时应该在主函数之前声明该函数的存在。声明时要在函数结尾处加一个分号。如果忘记加分号会导致编译失败。为了避免此类问题发生，最好把函数的定义直接放在主函数之前，省去声明的步骤，安全简便。

7.9.3　尽量不要用 using namespace std

尽量不要用 using namespace std，尽管这样很简便。using namespace std 会把标准命名空间里的东西都释放出来，如标准函数库、对象等。这样很容易造成标准库函数和自定义函数重名，导致错误。例如：



```
void swap(int a,int b)                         //swap 与标准库函数命名冲突，编译出错
{…}
swap(a,b);                                     //调用失败
```



Note

所以，最好是用到什么就释放什么，如“using std::cout;”。

7.10 本 章 小 结

本章主要介绍函数的使用，使用函数要了解函数的返回值、函数的参数以及函数的调用方式。变量的作用域和函数有关，函数的递归调用可以帮助开发人员设计出思路明了的程序，函数重载则解决了代码复用中函数名冲突的问题。

7.11 跟 我 上 机

参考答案：光盘\MR\跟我上机

使用冒泡法排序。设计一个函数，实现将数组 a 中的无序数按照升序重新排列并输出显示。

冒泡法排序：所有数两两比较，大的向后交换。N 个数，比较 N−1 轮，每比较一轮，筛选出一个数，完成排序。实现如下：

```
#include <iostream>
using namespace std;
#define N 5                                    //数组大小
void bubble(int *a)                            //排序函数
{
    int t = 0;                                 //定义中转变量
    for(int i = 0;i < N-1;i++)                 //比较 N−1 轮
        for(int j = 0;j < N-i-1;j++)           //两两比较
        {
            if(a[j] > a[j+1])                  //大的向后交换
            {
                t = a[j];
                a[j] = a[j+1];
                a[j+1] = t;
            }
        }
}
void print(int *a)                             //逐个输出数组元素
{
    for(int i = 0;i < N;i++)
        cout<<a[i]<<" ";
    cout<<endl;
}
```



```
void inPut(int *a)                                        //输入要比较的数
{
        int x = 1;
        for(int i = 0;i < N;i++)
        {
                cin>>a[i];
                cout<<"输入了"<<x++<<"个数\n";
        }
}
int main()
{
        int a[N] = {0};                              //定义整型数组 a
        cout<<"请输入要排序的数：\n";
        inPut(a);                                    //输入
        cout<<"\n 输入的"<<N<<"个数为：\n";
        print(a);                                    //输出排序前
        bubble(a);                                   //排序
        cout<<"排序后：\n";
        print(a);                                    //输出排序后
        return 0;
}
```



第8章

C++中的指针

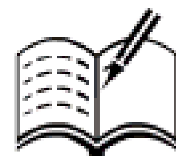
（视频讲解：20分钟）

程序中的所有变量与函数都存放到内存中，C++提供了指针对它们在内存中的地址进行访问。指针的功能强大，操作灵活，还能提高程序的运行效率。相对地，对内存操作是一把双刃剑，如果处理不当，将会造成程序的崩溃。

本章能够完成的主要范例（已掌握的在方框中打勾）

- [ ] 输出变量的地址
- [ ] 使用指针比较两个数的大小
- [ ] 指针的运算
- [ ] 使用指针作为返回值
- [ ] 动态分配内存空间
- [ ] 使用指针作为函数参数





8.1　指针概述

8.1.1　保存变量地址

系统的内存就像是带有编号的小房间，如果想使用内存就需要得到房间号。如图 8.1 所示，定义一个整型变量 i，它需要 4 个字节，所以编译器为变量 i 分配了编号从 4001 到 4004 的房间，每个房间代表一个字节。

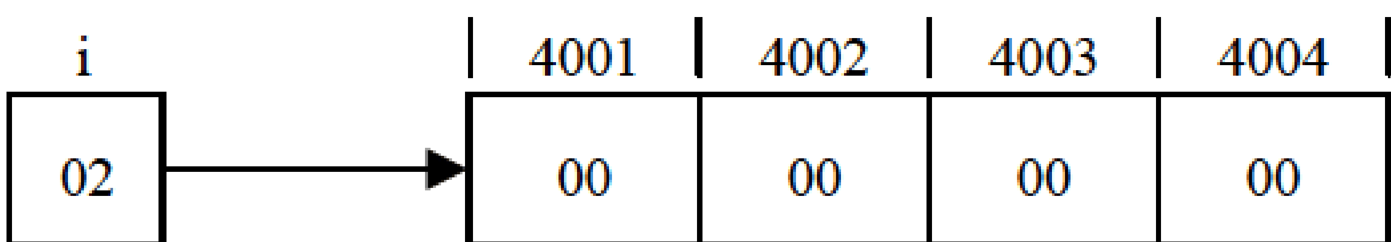


图 8.1　整型变量 i

各个变量连续地存储在系统的内存中，如图 8.2 所示，两个整型变量 i 和 j 存储在内存中。

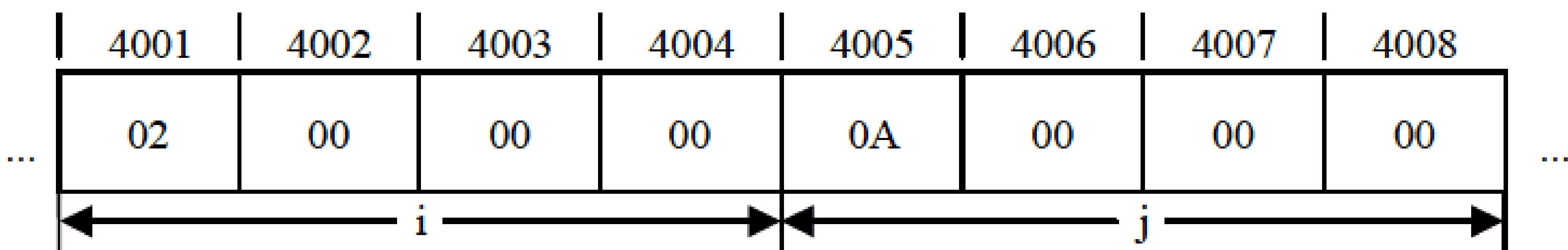


图 8.2　整型变量 i 和 j

在程序代码中是通过变量名来对内存单元进行存取操作，但是代码经过编译后已经将变量名转换为该变量在内存的存放地址，对变量值的存取都是通过地址进行的。例如，语句“i+j;”的执行过程是根据变量名与地址的对应关系，找到变量 i 的地址 4001，然后从 4001 开始读取 4 个字节数据放到 CPU 寄存器中，再找到变量 j 的地址 4005，从 4005 开始读取 4 个字节的数据放到 CPU 另一个寄存器中，通过 CPU 的加法中断计算出结果。

在低级语言的汇编语言中都是直接通过地址来访问内存单元的，而在高级语言中才使用变量名访问内存单元，C 语言作为高级语言却提供了通过地址来访问内存单元的方法，C++也继承了这一特性。

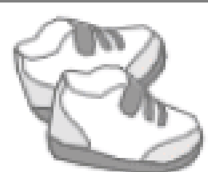
由于通过地址能访问指定的内存存储单元，可以说地址“指向”该内存单元，例如，房间号 4001 指向系统内存中的一个字节。地址可以形象地称为指针，意思是通过指针能找到内存单元。一个变量的地址称为该变量的指针。如果有一个变量专门用来存放另一个变量的地址，它就是指针变量。在 C++语言中有专门用来存放内存单元地址的变量类型，就是指针类型。

指针是一种数据类型，通常所说的指针就是指针变量，它是一个专门用来存放地址的变量，而变量的指针主要指变量在内存中的地址。变量的地址在编写代码时无法获取，只有在程序运行时才可以得到。

（1）指针的声明

声明指针的一般形式如下：

```
数据类型标识符 *指针变量名
```



例如：

```
int *p_iPoint;              //声明一个整型指针
float *a,*b;                //声明两个浮点指针
```



Note

（2）指针的赋值

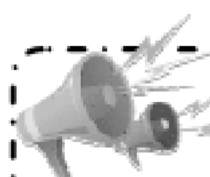
指针可以在声明时赋值，也可以后期赋值。

☑ 在初始化时赋值

```
int i = 100;
int *pPoint = &i;
```

☑ 在后期赋值

```
int i = 100;
int *pPoint;
pPoint = &i;
```



注意：

通过变量名访问一个变量是直接的，而通过指针访问一个变量是间接的。

（3）关于指针使用的说明

☑ 指针变量名是 p，而不是*p。

☑ p=&i 的意思是取变量 i 的地址赋给指针变量 p。

下面的实例可以获取变量的地址，并将获取的地址输出出来。

【例 8.1】 输出变量的地址值。

实例位置：光盘\MR\Instance\08\8.1

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
    int a=100;              //定义一个变量 a
    int *p=&a;              //定义一个指针变量 p 并初始化
    printf("%d\n",p);       //按十进制输出 a 的地址
}
```

程序运行结果如图 8.3 所示。

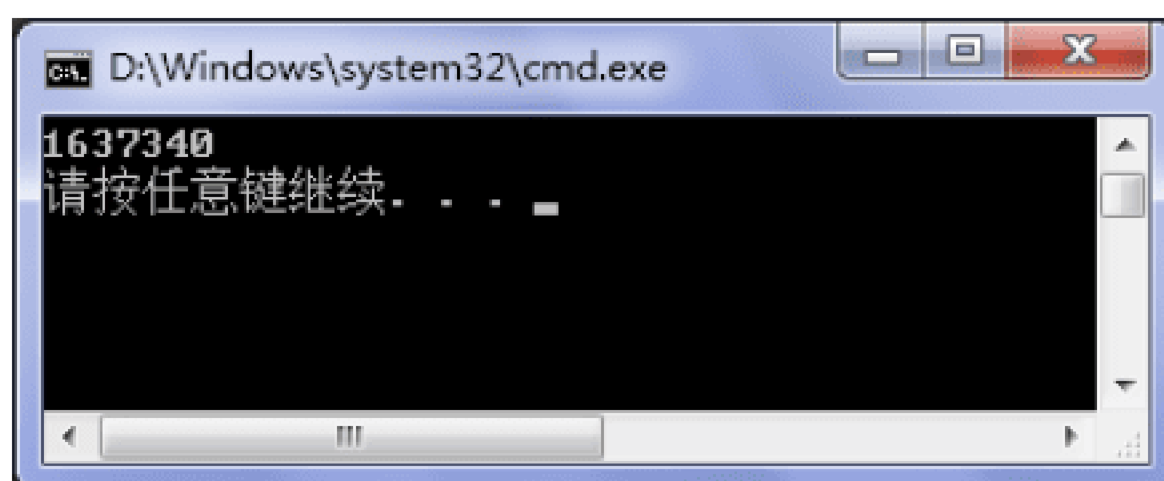


图 8.3　输出变量的地址值



实例可以通过 printf 函数直接将地址值输出出来。由于变量是由系统分配空间，所以变量的地址不是固定不变的。



注意：

在定义一个指针之后，一般要使指针有明确的指向。与常规的变量未赋值相同，没有明确指向的指针不会引起编译器出错，但是对于指针则可能导致无法预料的或者隐藏的灾难性后果，所以指针一定要赋值。

☑ 指针变量不可以直接赋普通常量的值。例如：

```
int a=100;
int *p;
p=100;                    //将常量 100 赋给指针变量
```

编译不能通过，有“error C2440: '=' : cannot convert from 'const int' to 'int *'”错误提示。

如果强行赋值，使用指针运算符*提取指针所指变量时会出错。例如：

```
int a=100;
int *p;
p=(int*)100;              //通过强制转换将 100 赋值给指针变量
printf("%d",p);           //输出地址，能够输出地址
printf("%d",*p);          //输出指针指向的值，出错语句
```

☑ 不能将*p 当变量使用。例如：

```
int a=100;
int *p;                   //指针未初始化
*p=100;                   //指针没有获得一个有效的地址
printf("%d",p);           //输出地址，出错语句
printf("%d",*p);          //输出指针指向的值，出错语句
```

上面代码可以编译通过，但运行时会弹出错误对话框，如图 8.4 所示。

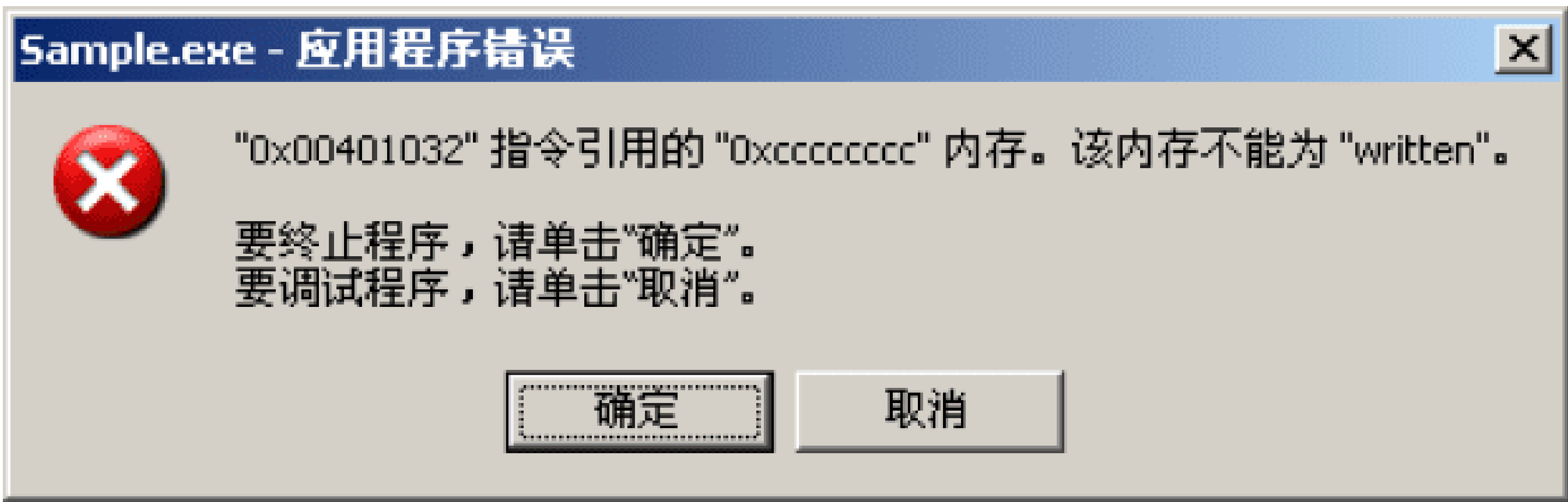
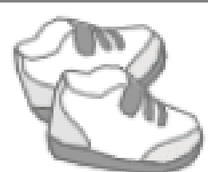


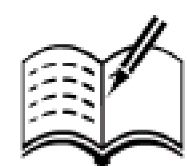
图 8.4　错误提示

（4）指针运算符和取地址运算符简介

*和&是两个运算符，*是指针运算符，&是取值运算符。

取地址运算符如图 8.5 所示，变量 i 的值为 100，存储在内存地址为 4009 的地方，取地址运算符&使指针变量 p 得到地址 4009。





指针运算符如图 8.6 所示，指针变量存储的是地址编号 4009，指针通过指针运算符可以得到地址。

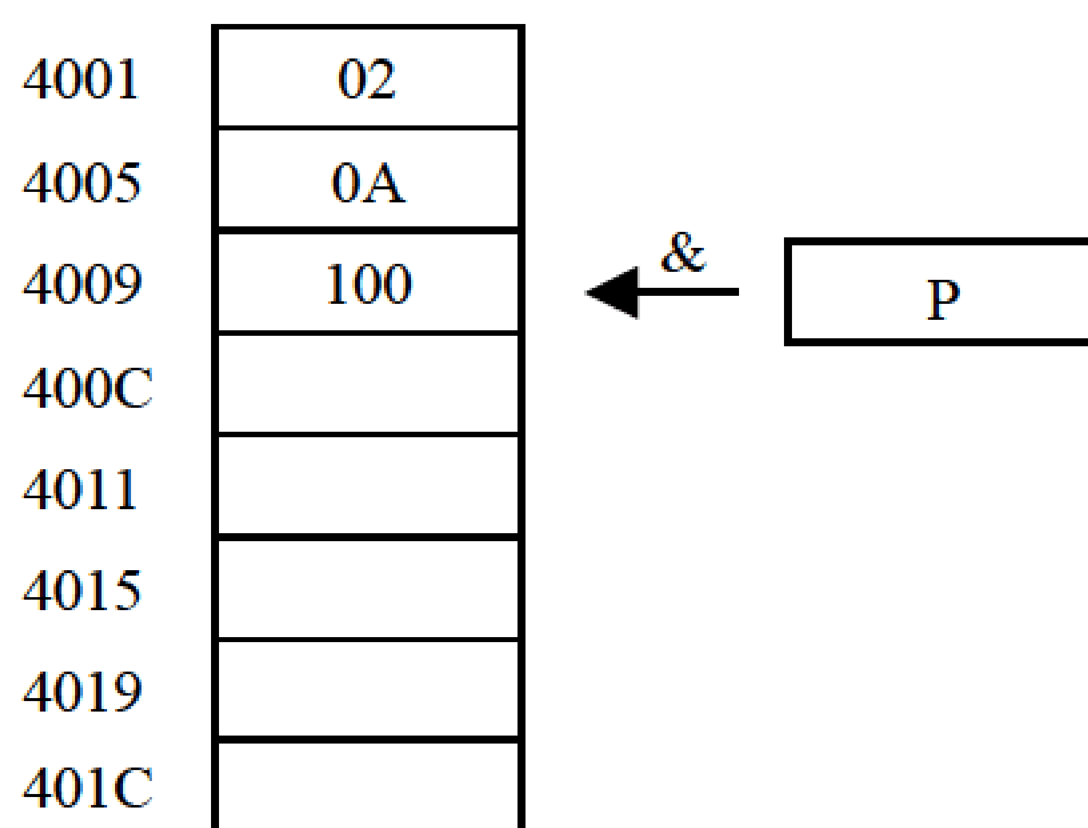


图 8.5　取地址

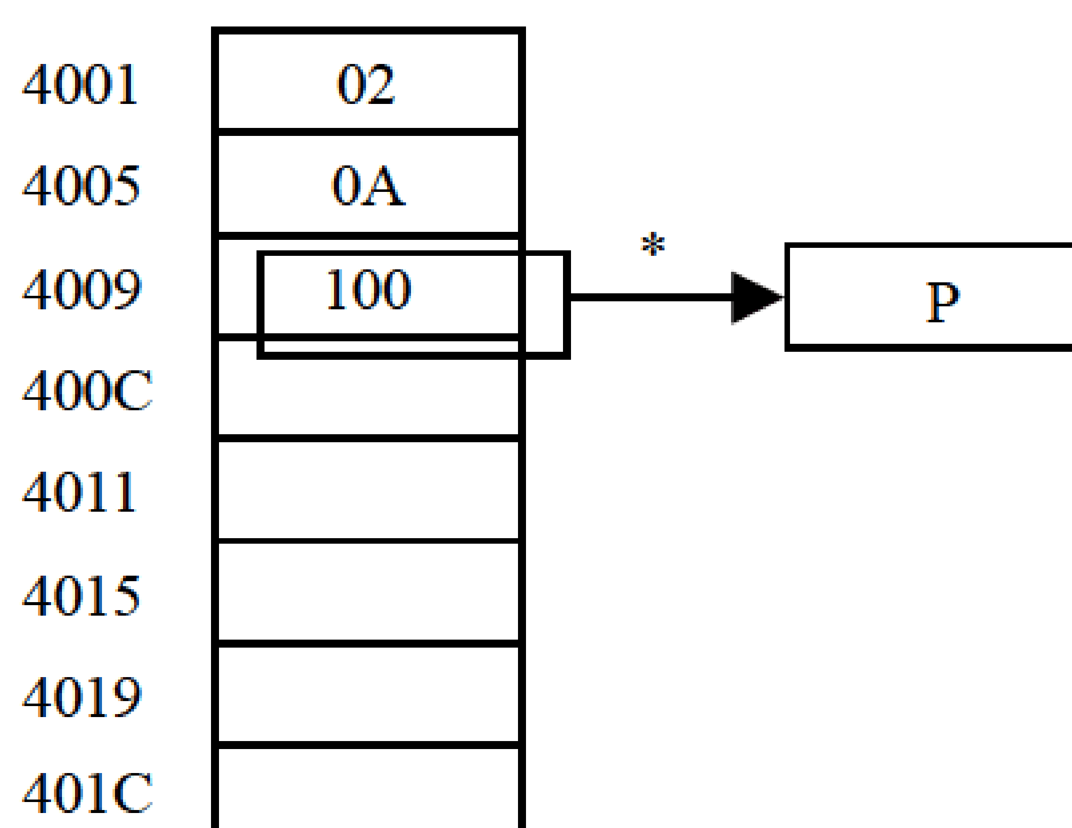


图 8.6　通过地址取值

下面实例通过指针来实现数据大小比较的功能。

【例 8.2】 使用指针比较两个数大小。

实例位置：光盘\MR\Instance\08\8.2

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int *p1,*p2;
	int	*p;				//临时指针
	int a,b;
	cout << "input a: " << endl;
	cin >> a;				//给 a 赋值
	cout << "input b: " << endl;
	cin >> b;
	p1=&a;p2=&b;			//p1 指向 a 地址，p2 指向 b 地址
	if(a<b)				//如果 a<b，交换 p1 和 p2 所指向的地址
	{
		p=p1;
		p1=p2;
		p2=p;
	}
	cout << "a=" << a;
	cout << " ";
	cout << "b=" << b;
	cout << endl;
	cout << "较大的数:" << *p1 << "较小的数:" << *p2 <<endl;
}
```

程序运行结果如图 8.7 所示。





图 8.7　使用指针比较两个数大小



（5）指针运算符和取地址运算符的说明

声明并初始化指针变量时同时用到了*和&这两个运算符。例如：

```
int *p=&a;
```

该语句等同于如下语句：

```
int * p;
p=&a;
```

如果写成“*p=&a;”，程序会报错。

&*p 中的 p 只能是指针变量，如果将*放在变量名前，编译时会有逻辑错误。例如：

```
#include <iostream>
using namespace std;
void main()
{
        int a=100;
        int *p;
        printf("%d",&*a);
}
```

编译程序会出现“error C2100: illegal indirection”的错误提示。

（6）&*p 和*&a 的区别

&和*的运算符优先级别相同，按自右而左的方向结合，因此，&*p 是先进行*运算，*p 相当于变量 a；再进行&运算，&*p 就相当于取变量 a 的地址。*&a 是先计算&运算符，&a 就是取变量 a 的地址，然后计算*运算，*&a 就相当于取变量 a 所在地址的值，实际就是变量 a。

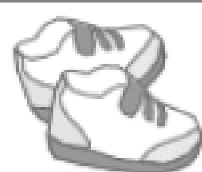
8.1.2　指针的运算

指针变量存储的是地址值，对指针做运算就等于对地址做运算。下面通过实例来使读者了解指针的运算。

【例 8.3】 输出 int 指针运算后的地址值。

实例位置：光盘\MR\Instance\08\8.3

```
#include "stdafx.h"
#include <iostream>
```





```
using namespace std;
void main()
{
	int a=100;
	int *p=&a;
	printf("address:%d\n",p);
	p++;
	printf("address:%d\n",p);
	p--;
	printf("address:%d\n",p);
	p--;
	printf("address:%d\n",p);
}
```

程序运行结果如图 8.8 所示。

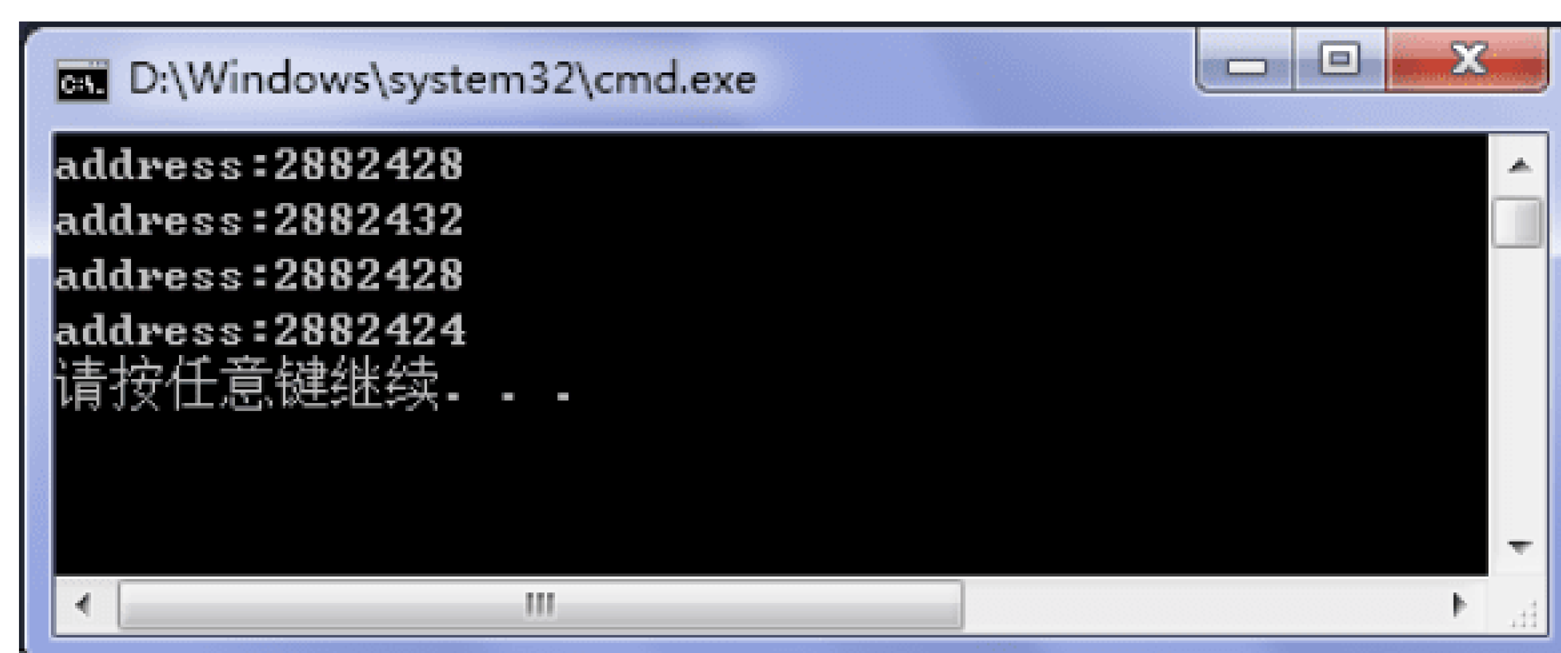


图 8.8 输出指针运算后地址值

程序首先输出的是指向变量 a 的指针地址值 1245052，然后对指针分别进行自加运算、自减运算、自减运算，输出的结果分别是 1245056、1245052、1245048。

指针进行一次加 1 运算，其地址值并没有加 1，而是增加了 4，这和声明指针的类型有关。

p++是对指针做自加运算，相当于语句 p=p+1，地址是按字节存放数据，但指针加 1 并不代表地址值加 1 个字节，而是加上指针数据类型所占的字节宽度，要获取字节宽度需要使用 sizeof 关键字，例如，整型的字节宽度是 sizeof(int)，sizeof(int)的值为 4。双精度整型的字节宽度是 sizeof(double)，其值为 8。将实例中的 int 指针类型改为 double，看看运行结果，代码如下。

【例 8.4】 输出 double 指针运算后的地址值。

实例位置：光盘\MR\Instance\08\8.4

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	double a=100;
	double *p=&a;
	printf("address:%d\n",p);
	p++;
	printf("address:%d\n",p);
	p--;
	printf("address:%d\n",p);
```




```
    p--;
    printf("address:%d\n",p);
}
```

程序运行结果如图 8.9 所示。

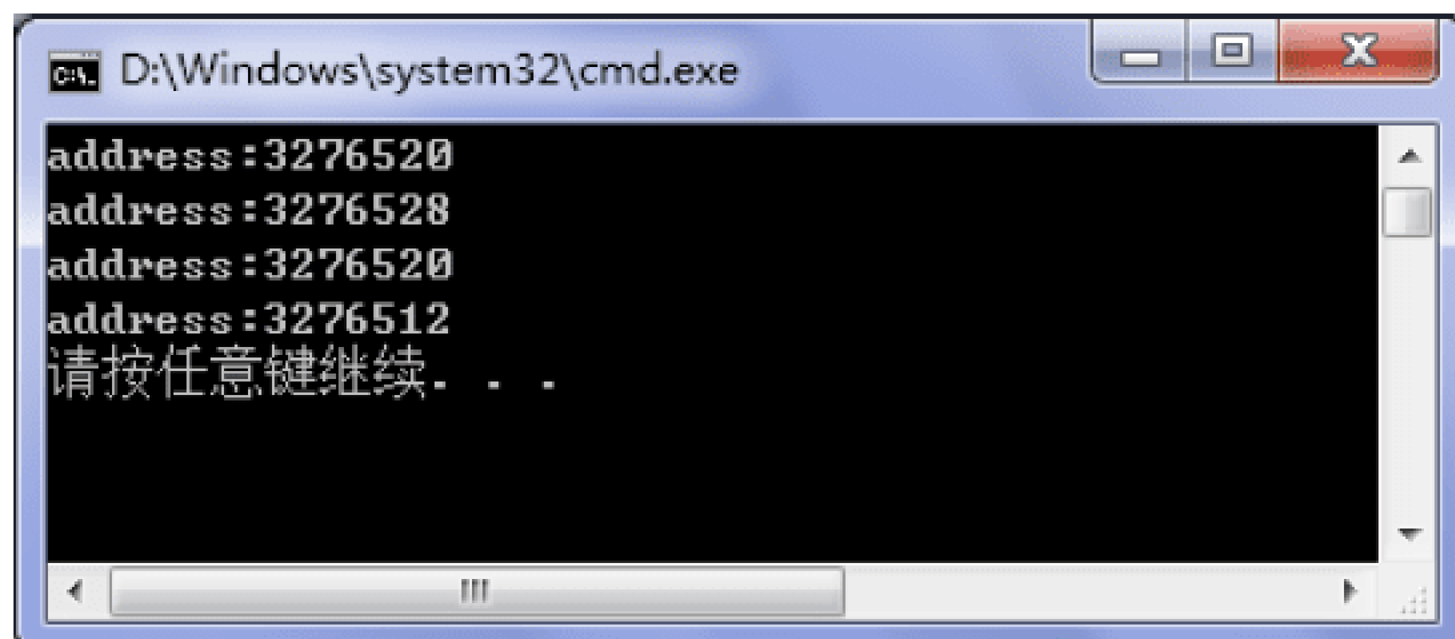
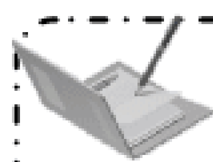


图 8.9　运行结果



说明：

定义指针变量时必须指定一个数据类型。指针变量的数据类型用来指定该指针变量所指向数据的类型。

8.1.3　指向空的指针与空类型指针

指针可以指向任何数据类型的数据，包括空类型（void）：

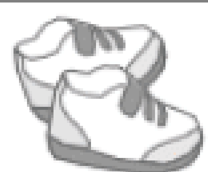
```
void* p;        //定义一个指向空类型的指针变量
```

空类型指针可以接收任何类型的数据，当使用它时，可以将其强制转换为所对应数据类型。

【例 8.5】 空类型指针的使用。

实例位置：光盘\MR\Instance\08\8.5

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main()
{
    int *pI = NULL;
    int i = 4;
    pI = &i;
    float f = 3.333f;
    bool b =true;
    void *pV = NULL;
    cout<<"依次赋值给空指针"<<endl;
    pV = pI;
    cout<<"pV = pI --------"<<*(int*)pV<<endl;
    cout<<"pV = pI ---------转为 float 类型指针"<<*(float*)pV<<endl;
    pV = &f;
```



```
    cout<<"pV = &f --------"<<*(float*)pV<<endl;
    cout<<"pV = &f --------转为 int 类型指针"<<*(int*)pV<<endl;
    return 0;
}
```



Note

程序运行结果如图 8.10 所示。

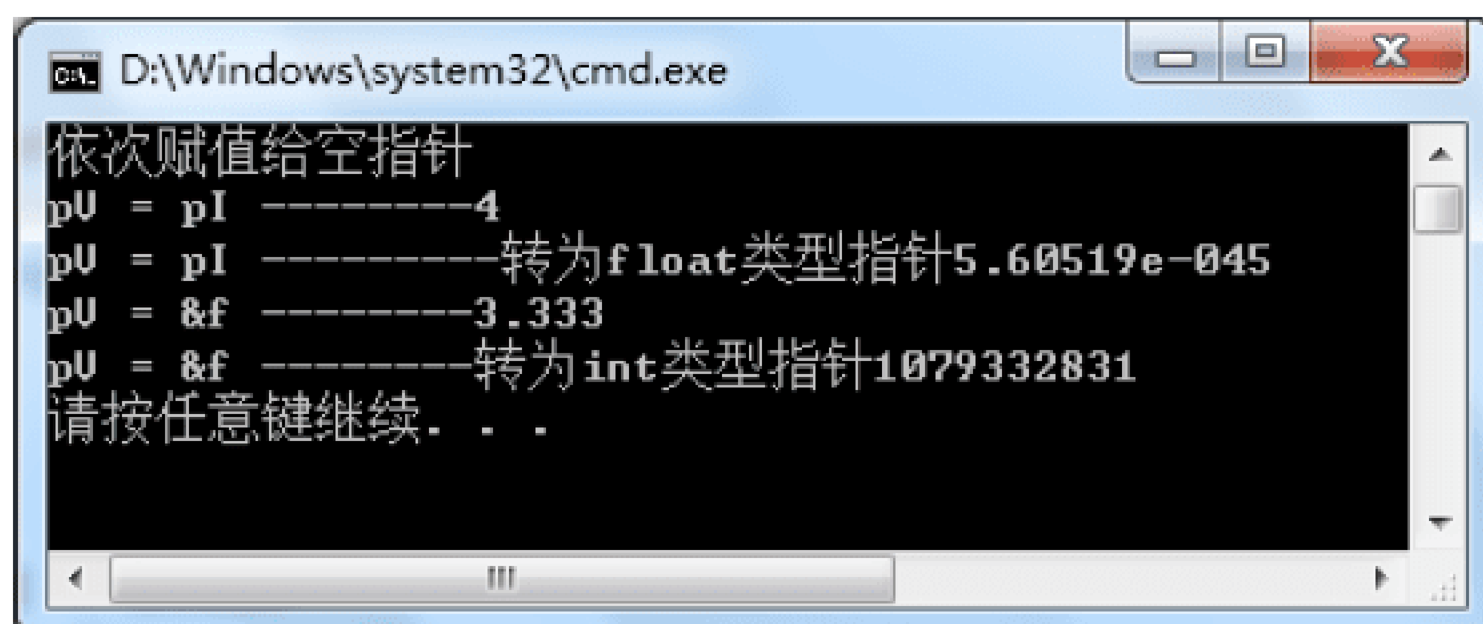


图 8.10　执行结果

可以看到空指针赋值后，转换为对应类型的指针才能得到所期望的结果。若将它转换为其他类型的指针，得到的结果将不可预知，非空类型指针同样具有这样的特性。在本实例中，出现了一个符号 NULL，它表示空值。空值无法用输出语句表示，而且赋空的指针无法被使用，直到它被赋予其他的值。

8.1.4　指向常量的指针与指针常量

同其他数据类型一样，指针也有常量，使用 const 关键字形式如下：

```
int i =9;
int * const p = &i;
*p = 3;
```

将关键字 const 放在标识符前，表示这个数据本身是常量，而数据类型是 int*，即整型指针。与其他常量一样，指针常量必须初始化。我们无法改变它的内存指向，但是可以改变它指向内存的内容。

若将关键字 const 放到指针类型的前方，形式如下：

```
int i =9;
int const* p = &i;
```

这是指向常量的指针，虽然它所指向的数据可以通过赋值语句进行修改，但是通过该指针修改内存内容的操作是不被允许的。

当 const 以如下形式使用时：

```
int i =9;
int const* const p = &i;
```

该指针是一个指向常量的指针常量。既不可以改变它的内存指向，也不可以通过它修改指向





内存的内容。

【例 8.6】 指针与 const。

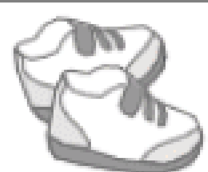
实例位置：光盘\MR\Instance\08\8.6

```
#include "stdafx.h"
#include <iostream>
using std::cout;
using std::endl;
int main()
{
	int i = 5;
	const int c = 99;
	const int* pR = &i;		//这个指针只能用来“读”内存数据，但可以改变自己的地址
	int* const pC = &i;		//这个指针本身是常量，不能改变指向，但它能够改变内存的内容
	const int* const pCR = &i;	//这个指针只能用来“读“内存数据，并且不能改变指向
	cout<<"三个指针都指向了同一个变量 i，同一块内存"<<endl;
	cout<<"指向常量的指针 pR 操作:"<<endl;
	//*pR = 6			//去掉语句前方注释报错
	cout<<"通过赋值语句修改 i:"<<endl;
	i = 100;
	cout<<"i:"<<i<<endl;
	cout<<"将 pR 的地址变成常量 c 的地址:"<<endl;
	pR = &c;
	cout<<"*pR:"<<*pR<<endl;
	cout<<"指向常量的指针 pC 操作:"<<endl;
	//pC = &c;			//去掉语句前方注释报错
	cout<<"通过 pC 改变 i 值:"<<endl;
	*pC = 6;
	cout<<"i:"<<i<<endl;
	cout<<"指向常量的指针常量 pCR 操作:"<<endl;
	//pCR =&c;			//报错
	//*pCR =100;			//报错
	cout<<"通过 pCR 无法改变任何东西，真正作到了只读"<<endl;
	return 0;
}
```

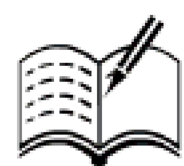
程序运行结果如图 8.11 所示。

```
D:\Windows\system32\cmd.exe
三个指针都指向了同一个变量i，同一块内存
指向常量的指针pR操作:
通过赋值语句修改i:
i:100
将pR的地址变成常量c的地址:
*pR:99
指向常量的指针pC操作:
通过pC改变i值:
i:6
指向常量的指针常量pCR操作:
通过pCR无法改变任何东西，真正作到了只读
请按任意键继续. . .
```

图 8.11　执行结果



8.2 指针在函数中的应用



Note

8.2.1 传递地址

以前所接触到的函数都是按值传递参数，也就是说实参传递进函数体内后，生成的是实参的副本。当在函数内改变副本的值时并不影响到实参。而指针传递参数时，指针变量产生了副本，但副本与原变量所指向的内存区域是同一个。对指针副本指向的变量进行改变，就是改变原指针变量所指向的变量。

【例 8.7】 调用自定义函数交换两变量值。

实例位置：光盘\MR\Instance\08\8.7

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void swap(int *a,int *b)                    //交换 a、b 指向的两个地址的值（指针传递）
{
    int tmp;                                //定义一个临时变量
    tmp=*a;                                 //把 a 指向的值赋给 tmp
    *a=*b;                                  //把 b 指向的值赋到 a 指向的位置
    *b=tmp;                                 //把 tmp 赋给 b 指向的位置
}
void swap(int a,int b)                      //交换 a、b 的值（值传递）
{
    int tmp;
    tmp=a;
    a=b;
    b=tmp;
}
void main()
{
    int x,y;
    int *p_x,*p_y;                          //定义两个整型指针
    cout << "input two number" << endl;
    cin >> x;                               //给 x、y 赋值
    cin >> y;
    p_x=&x;p_y=&y;                          //两个指针分别指向 x、y 的地址
    cout<<"按指针传递参数交换"<<endl;
    swap(p_x,p_y);                          //执行的是参数列表都为指针的 swap 函数
    cout << "x=" << x <<endl;
    cout << "y=" << y <<endl;
    cout<<"按值传递参数交换"<<endl;
    swap(x,y);                              //执行的是参数列表为整型变量的 swap 函数
    cout << "x=" << x <<endl;
```





Note

```
    cout << "y=" << y <<endl;
}
```

程序运行结果如图 8.12 所示。

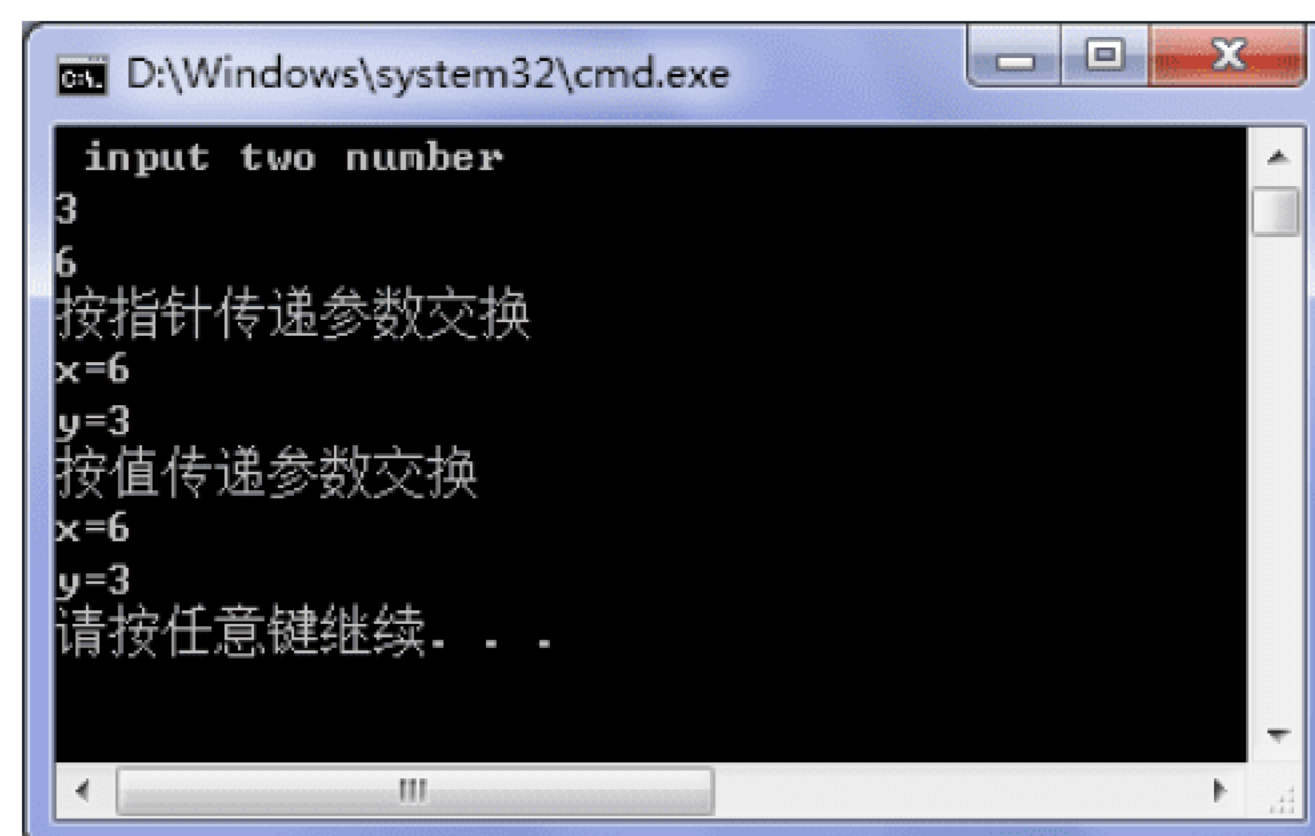


图 8.12　调用自定义函数交换两变量值

从图 8.12 的结果中可以看出，使用指针传递参数的函数真正实现了 x 与 y 的交换，而按值传递函数只是交换了 x 与 y 的副本。

swap 函数是用户自定义的重载函数，在 main 函数中调用该函数交换变量 a 和 b 的值。按指针传参的 swap 函数的两个形参 a、b 被传入了两个地址值，也就是传入了两个指针变量，在 swap 函数的函数体内使用整型变量 tmp 作为中转变量，将两个指针变量所指向的数值进行交换。在 main 函数内首先获取输入的两个数值，分别传递给变量 x 和 y，将 x 和 y 的地址值传递给 swap 函数。在按指针传递的 swap 函数内，两个指针变量的副本 a 和 b 所指向的变量正是 x 与 y。而按值传递的 swap 函数并没有实现交换 x 与 y 的功能。

8.2.2　指向函数入口地址

指向函数入口地址的指针被称为函数指针。一个函数在编译时被分配给一个入口地址，这个函数入口地址就称为函数的指针。可以用一个指针变量保存函数的入口地址，然后通过该指针变量调用此函数。

一个函数可以返回一个整型值、字符值、实型值等，也可以返回指针型的数据，即地址。其概念与以前类似，只是返回的值的类型是指针类型而已。返回指针值的函数简称为指针函数。

定义指针函数的一般形式为：

```
类型名 *函数名(参数表列);
```

例如，定义一个具有两个参数和一个返回值的函数指针和一个具有同样返回值参数列表的函数：

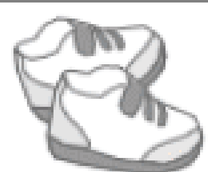
```
int sum(int x,int y)                //定义一个函数
int *a(int ,int );                  //定义一个函数指针
a = sum;                            //让函数指针 a 指向函数 sum
```

函数指针能指向返回值与参数列表的函数，当使用函数指针时形式如下：

```
int c,d;                            //定义两个整型变量
(*a)(c ,d );                        //调用指针 a 指向的函数，并传参
```

下面定义通过函数指针实现求和与求平均值的计算。

【例 8.8】 使用指针函数进行计算。





实例位置：光盘\MR\Instance\08\8.8

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int avg(int a,int b);                          //声明函数 avg
int sum(int a,int b);                          //声明函数 sum
void main()
{
    int iWidth,iLenght,iResult;
    iWidth = 10;
    iLenght = 30;
    int (*pFun)(int,int);                      //定义函数指针
    cout << "pFun 指向了 avg"<<endl;
    pFun = avg;                                //让 pFun 指向函数 avg
    iResult=(*pFun)(iWidth,iLenght);           //调用函数 avg
    cout <<"执行结果:"<< iResult <<endl;
    cout << "pFun 指向了 sum"<<endl;
    pFun = sum;                                //让指针指向 sum
    iResult=(*pFun)(iWidth,iLenght);           //调用 sum 函数
    cout <<"执行结果:"<< iResult <<endl;
}
int sum(int a,int b)                           //求和函数
{
    return a+b;
}
int avg(int a,int b)                           //求平均数函数
{
    return (a+b)/2;
}
```

程序运行结果如图 8.13 所示。

pFun 函数指针先后指向了平均值函数、求和函数。同过对它自身指向的函数调用，得到了各自的结果。



图 8.13　执行结果

8.2.3　空指针调用函数

空类型指针指向任意类型函数或者将任意类型的函数指针赋值给空类型指针都是合法的。使用空指针调用自身所指向的函数，仍然按照强制转换的形式使用。

【例 8.9】 使用空类型指针执行函数。

实例位置：光盘\MR\Instance\08\8.9

```
#include "stdafx.h"
#include <iostream>
```






```
using std::cin;
using std::cout;
using std::endl;
int plus(int b)
{
	return b+1;
}
int main()
{
	void* pV = NULL;                    //定义一个空类型指针
	int result = 0;
	pV = plus;                          //让指针指向函数 plus
	cout<<"执行 pV 指向的函数:"<<endl;
	result=((int(*)(int))pV)(10);       //将空类型指针 pV 强制转换返回值整型、参数整型的函数指针
	cout<<"result:"<<result<<endl;
	return 0;
}
```

程序运行结果如图 8.14 所示。



图 8.14　执行结果



注意：

当函数被重载时，不要使用直接将函数名赋给空类型指针的操作（如例 8.9），这会使编译器无法确定将哪个重载函数交给空类型指针。

8.2.4　从函数中返回指针

定义一个返回指针类型的函数，形式如下：

```
int* function(参数列表)
{
	…;//执行过程
	return   p;
}
```

p 是一个指针变量，也可以是形式如&value 的地址值。当函数返回一个指针变量时，得到的是地址值。值得注意的是，返回指针的内存内容并不随返回的地址一样经过复制成为临时变量。





如果操作不当，后果将难以预料。

【例 8.10】 指针做返回值。

实例位置：光盘\MR\Instance\08\8.10

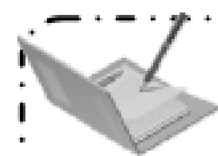
```
#include "stdafx.h"
#include <iostream>
using std::cout;
using std::endl;
int* pointerGet(int* p)
{
	int i = 9;
	cout<<"函数体中 i 的地址"<<&i<<endl;
	cout<<"函数体中 i 的值:"<<i<<endl;
	p = &i;
	return p;
}
int main()
{
	int* k = NULL;
	cout<<"k 的地址:"<<k<<endl;			//输出 k 的初始地址
	cout<<"执行函数，将 k 赋予函数返回值"<<endl;
	k = pointerGet(k);					//调用函数获得一个指向变量 i 的地址的指针
	cout<<"k 的地址:"<<k<<endl;			//输出 k 的新地址（i 的地址）
	cout<<"k 所指向内存的内容:"<<*k<<endl;	//输出一个随机数
}
```

程序运行结果如图 8.15 所示。



图 8.15 执行结果

可以看到，函数返回的是函数中定义的 i 的地址。函数执行后，i 的内存被销毁，值变成了一个不可预知的数。



说明：

值为 NULL 的指针地址为 0，但并不意味着这块内存可以使用。将指针赋值为 NULL 也是基于安全而考虑的，以后的章节我们还将详细讨论内存的安全问题。





8.3　安全使用指针

8.3.1　内存分配

1. 堆与栈

在程序中定义一个变量，它的值会被放入内存当中。如果没有申请动态分配的方式，它的值将放到栈中。在栈中的变量所属的内存大小是无法被改变的，它们的产生与消亡也与变量定义的位置和储存方式有关。与栈相对应的，堆是一种动态分配方式的内存。当申请使用动态分配方式去储存某个变量时，这个变量会被放入堆中。根据需要，这个变量的内存大小可以发生改变，内存的申请和销毁的时机则由编程者来操作。

2. 关键字 new 与 delete

创建变量之前，编译器没有获取到变量的名称，只具有指向该变量的指针。那么，申请变量的堆内存即是申请自身指向堆。new 是 C++语言申请动态内存的关键字，形式如下：

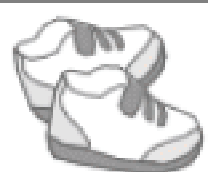
```
p1 = new type;
```

其中，p1 表示指针，new 是关键字，type 是类型名。new 返回新分配的内存单元的地址。这样，p1 指针就申请了动态方式，使用它在堆内申请的内存储存 int 类型的值。

【例 8.11】 动态分配空间。

实例位置：光盘\MR\Instance\08\8.11

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main()
{
	int* pl1 = NULL;
	pl1 = new int;				//申请动态分配
	*pl1 = 111;				//动态分配的内存储存的内容变成 111 的整型变量
	cout<<"pl1 内存的内容"<<*pl1<<",pl1 所指向的地址"<<pl1<<endl;
	int* pl2;
	//*pl2 = 222;				//直接赋值会导致错误!!!
	int k ;					//栈中的变量
	pl2 = &k;				//分配栈内存
	*pl2 = 222;				//分配内存后方可赋值
	cout<<"pl2 内存的内容"<<*pl2<<",pl2 所指向的地址"<<pl2<<endl;
	return 0;
}
```



可以看到指针 pI1 创建后申请了动态分配，程序自动交给了它一块堆内存。而指针 pI2 则是获取了栈中的内存地址，属于静态分配。



Note

程序运行结果如图 8.16 所示。

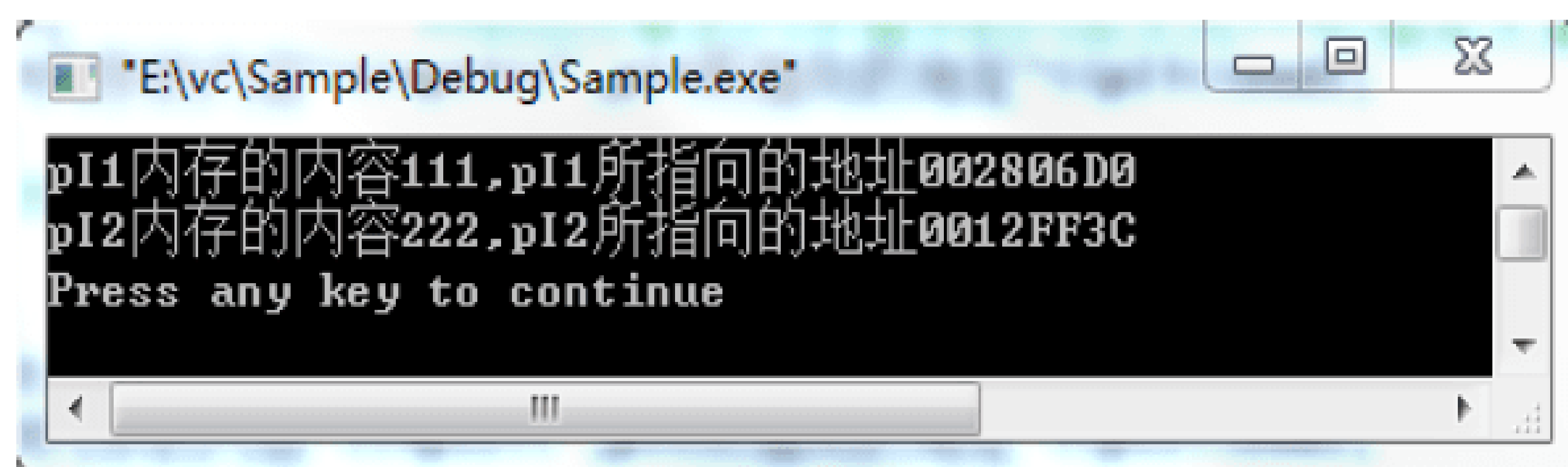


图 8.16　执行结果

动态分配方式虽然很灵活，但是随之带来新的问题：申请一块堆内存后，系统不会在程序执行时依据情况自动销毁它。若想释放该内存空间，则需要使用 delete 关键字。

【例 8.12】 动态内存的销毁。

实例位置：光盘\MR\Instance\08\8.12

```
#include "stdafx.h"
#include <iostream>
using std::cout;
using std::endl;
int* newPointerGet(int* p1)
{
	int k1 = 55;
	p1 =new int;			//变为堆内存
	* p1 = k1;			//int 型变量赋值操作
	return p1;
}
int* PointerGet(int *p2)
{
	int k2 = 55;
	p2 =&k2;			//指向函数中定义变量所在的栈内存，此段内存在函数执行后销毁
	return p2;
}

int main()
{
	cout<<"输出函数各自返回指针所指向的内存的值"<<endl;
	int* p =NULL;
	p = newPointerGet(p);		//p 具有堆内存的地址
	int* i=NULL;
	i = PointerGet(i);		//i 具有栈内存地址，内存内容被销毁
	cout<<"newGet: "<<*p<<" , get: "<<*i<<endl;
	cout<<"i 所指向的内存没有被立刻销毁，执行一个输出语句后:"<<endl;
					//i 仍然为 55，但不代表程序不会对它进行销毁
	cout<<"newGet: "<<*p<<" , get: "<<*i<<endl;	//执行其他的语句后，程序销毁了栈空间
	delete p;			//依照 p 销毁堆内存
	cout<<"销毁堆内存后:"<<endl;
	cout<<"*p:	"<<*p<<endl;
```





Note

```
	return 0;
}
```

程序运行结果如图 8.17 所示。

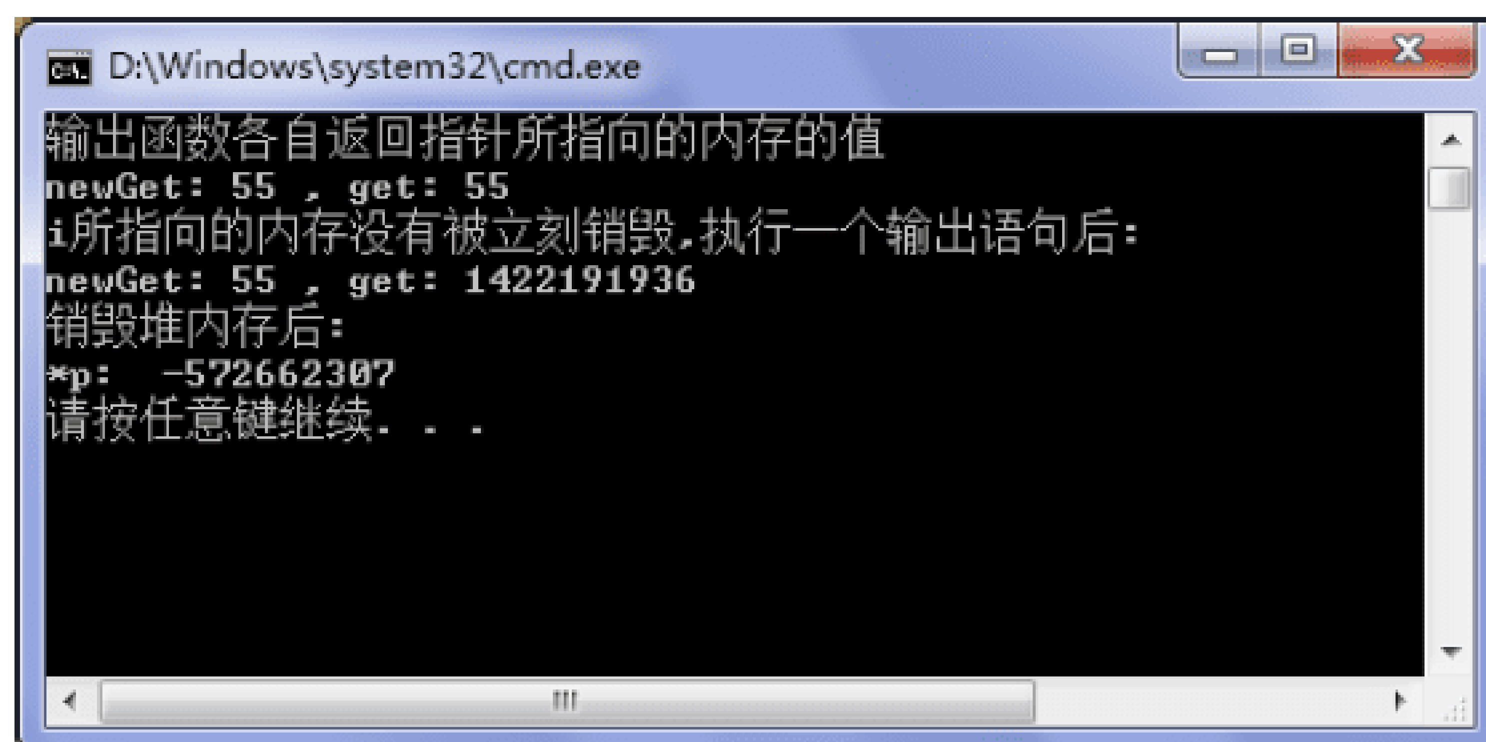


图 8.17　执行结果

变量 p 接收了 newGet 返回的指针的堆内存地址，所以内存的内容并没被销毁，而栈内存则由系统控制。程序最后使用 delete 语句释放了堆内存。

8.3.2　内存安全

指针是 C++提供给我们的强大而灵活的工具，如何安全地使用它们对内存安全的操作是编程者必须要掌握的。在前面的章节中讨论过指针所指向内存销毁的问题，当一块内存被销毁时，该区域不可复用。若有指针指向该区域，则需要将该指针置空值（NULL）或者指向未被销毁的内存。

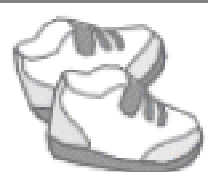
内存销毁实质上是系统判定该内存不是编程人员正常使用的空间，系统也会将它分配给别的任务。若擅自使用被销毁内存的指针更改该内存的数据，很可能会造成意想不到的结果。

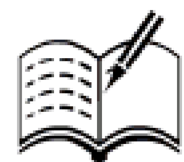
【例 8.13】 被销毁的内存。

实例位置：光盘\MR\Instance\08\8.13

这是一个反例，也许它会造成内存出错。

```
#include "stdafx.h"
#include <iostream>
using std::cout;
using std::endl;
int* sum(int a,int b)
{
	int* pS =NULL;
	int c = a+b;
	pS = &c;
	return pS;
}
int main()
{
	int* pI = NULL;			//将指针初始化为空
```






```
        int k1 = 3;
        int k2 = 5;
        pl = sum(k1,k2);
        cout<<"*pl 的值:"<<*pl<<endl;
        cout<<"也许*pl 还保留着 i 值，但它已经被程序认定为销毁"<<endl;
        cout<<"*pl 的值:"<<*pl<<endl;
        cout<<"尝试修改*pl"<<endl;
        *pl = 3;
        for(int i= 0;i<3;i++)
        {
            cout<<"修改被销毁的内存后*pl 的值:"<<*pl<<endl;
        }
}
```

程序运行结果如图 8.18 所示。



图 8.18　执行结果

指针 pI 从 sum 函数中得到了一个临时指针，该指针是指针 pS 的临时复制品，操作完成后消失，它所保留的地址交给了 pI。在函数 sum 执行完毕后，该域使用的栈内存会被系统销毁甚至挪用。本程序尝试通过 pI 继续使用、修改它，结果是系统会再次销毁它。在某些场合下，该程序也许会引起内存报错，甚至造成多个程序崩溃。所以对于栈内存的指针一定要明白其何时销毁，不再重复利用它。

与此相对应的另一个安全问题叫内存泄漏。如同我们所知道的，在申请动态分配内存后，系统不会主动销毁该堆内存，需要编程者使用 delete 关键字通知系统销毁。如果不这样做，系统将浪费很多资源，使程序执行时变得臃肿，只需占用数十 MB 内存的程序可能为此占用上百 MB 的内存。可见，回收堆内存空间是很重要的。销毁内存时，需要保留指向该堆内存的指针。当没有指针指向一块没被回收的堆内存时，此块内存犹如丢失了一般，我们称为内存泄漏。

【例 8.14】 丢失的内存。

实例位置：光盘\MR\Instance\08\8.14

这是一个反例，它会造成内存泄漏。

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main()
{
        float* pF = NULL;
        pF = new float;                    //动态申请一块内存，用 pF 去指向
        *pF = 4.321f;
        float f2 = 5.321f;
        cout<<"pF 指向的地址:"<<pF<<endl;
        cout<<"*pF 的值:"<<*pF<<endl;
        pF = &f2;                          //让 pF 指向了另一地址，此时上面申请的内存变为不可用
```






```
        cout<<"pF 指向了 f2 的地址:"<<pF<<endl;
        if(*pF>5)
        {
                cout<<"*pF 的值:"<<*pF<<endl;
        }
        return 0;
}
```

程序运行结果如图 8.19 所示。

程序中动态分配的内存开始由*pF 指向。当 pF 改变指向后，此块内存再也无法回收了。

一般情况下，我们无法通过调试程序发现内存泄漏。所以，使用动态分配时一定要注意形成良好的习惯。

D:\Windows\system32\cmd.exe
pF指向的地址:001F76C8
*pF的值:4.321
pF指向了f2的地址:0013F954
*pF的值:5.321
请按任意键继续. . .

图 8.19　执行结果

【例 8.15】 回收动态内存的一般处理步骤。

实例位置：光盘\MR\Instance\08\8.15

```
#include "stdafx.h"
#include <iostream>
void swap(int* a,int* b)
{
        int temp = *a;
        *a = *b;
        *b = temp;
}
int main()
{
        int* pI =new int;
        *pI = 3;
        int k =5;
        swap(pI,&k);
        std::cout<<"*pI:"<<*pI<<std::endl;      //使用 std 名字空间
        std::cout<<"k:"<<k<<std::endl;
        delete pI ;                             //回收动态内存
        pI = NULL;                              //将 pI 置空，防止使用已销毁的内存
                                                //和上一语句不可颠倒，否则将造成内存泄漏
        return 0;
}
```

程序运行结果如图 8.20 所示。

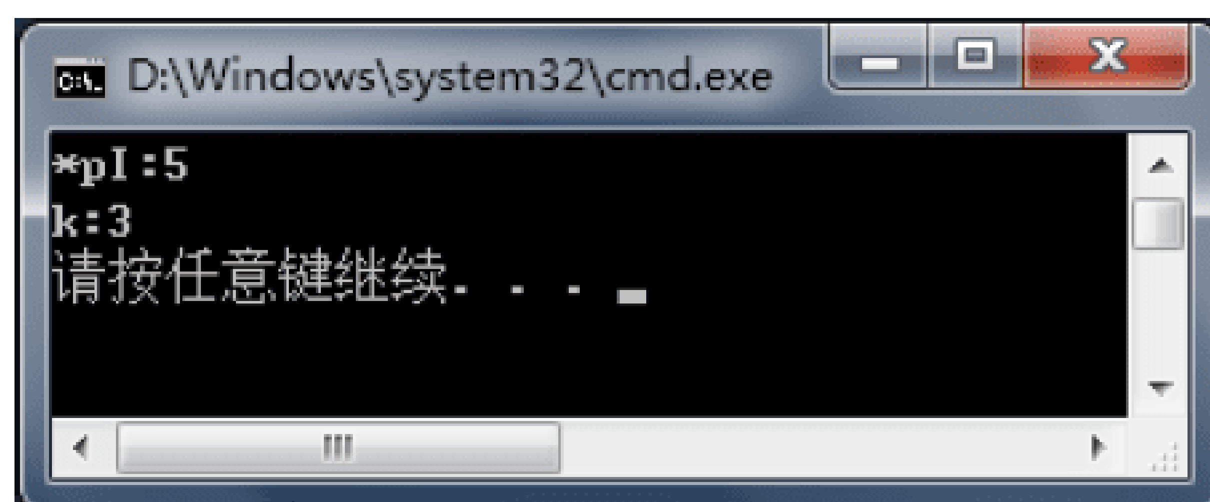
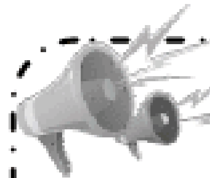


图 8.20　执行结果





注意：

指针是一种灵活高效的内存访问机制，它可以通过变量在内存中的地址来对变量直接操作。但是指针却不能访问寄存器变量，因为寄存器变量并没有保存在内存中，而是保存在寄存器中（从寄存器中读取数据要比从内存中读取数据速度快，所以有些要求频繁使用的数据可以被放在寄存器中），指针只能访问内存，不能访问寄存器，所以指针访问不到寄存器变量。



Note

8.4 综合应用

8.4.1 水桶的平衡

【例 8.16】 现在有两桶带有 100 个刻度的水桶，分别向它们的内部装水，当它们的水位达到相同刻度时，求出交换水量。本实例使用指针设计一个函数来解决这个问题。主要代码如下：

实例位置：光盘\MR\Instance\08\8.16

```
#include "stdafx.h"
#include <iostream>
using namespace std;
//平衡水量的函数。平衡两桶水可以先取出两个水量的差值部分，再平均分给两个桶
int balance(int *pA,int *pB)                    //定义一个平衡函数，参数是两个桶原来各自的水量
{
	int offset = *pA - *pB;                     //求出两个水量的差值
	*pA -= offset/2;                            //将水量差分为两份，一份分给 a
	*pB += offset/2;                            //将水量差分为两份，一份分给 b
	if(*pA == *pB)                              //水位平衡
		return offset;                          //返回交换的水量
}
int main(int argc, _TCHAR* argv[])
{
	int stockA;
	int stockB;
	while(true)
	{
		cout<<"输入第一个桶的水量:"<<endl;
		cin>>stockA;
		cout<<"输入第二个桶的水量:"<<endl;
		cin>>stockB;
		if(stockA>=0&&stockA<=100&&stockB>=0&&stockB<=100)
		{
			break;
		}
		else
		{
			cout<<"您的输入有误，桶的刻度范围为 0-100"<<endl;
		}
	}
```





```
        int offset = balance(&stockA,&stockB);
        if(offset>0)
        {
                cout<<"第一桶水比第二桶水多了"<<offset<<"刻度"<<endl;
        }
        else    if(offset<0)
        {
                cout<<"第一桶水比第二桶水少了"<<-offset<<"刻度"<<endl;
        }
        else
        {
                cout<<"第一桶水和第二桶水的刻度一致"<<endl;
        }
        return 0;
}
```

程序运行结果如图 8.21 所示。

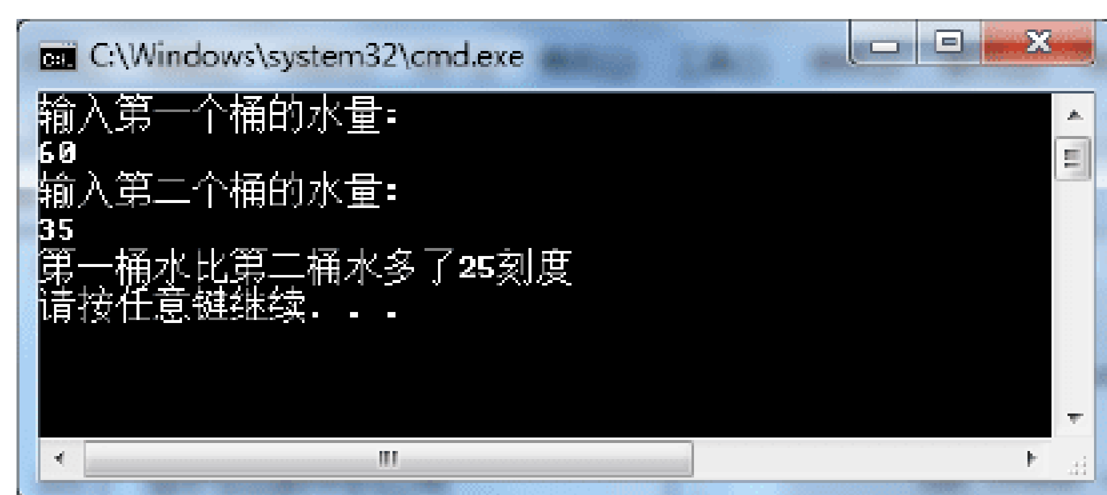


图 8.21　水桶的平衡

8.4.2　分步计算

【例 8.17】 求两个整数的平方和是一道数学计算题，通过本章知识讲解，可以设计两个函数来计算出结果，一个函数计算整数的平方，另一个函数计算两个数的和，运用指针可以提高计算效率。

程序主要代码如下：

实例位置：光盘\MR\Instance\08\8.17

```
int square(const int &x)                    //使用左值引用不进行复制
{
        return x*x;                         //求平方
}
int add(int &&p,int &&temp)                 //使用右值引用提高效率
{
        return p+temp;                      //求和
}
```

主函数中调用函数的形式为：

```
cout<<"平方和为"<<add(square(dV1),square(dV2))<<endl;    //返回值作参数（链式操作）
```

程序运行效果如图 8.22 所示。

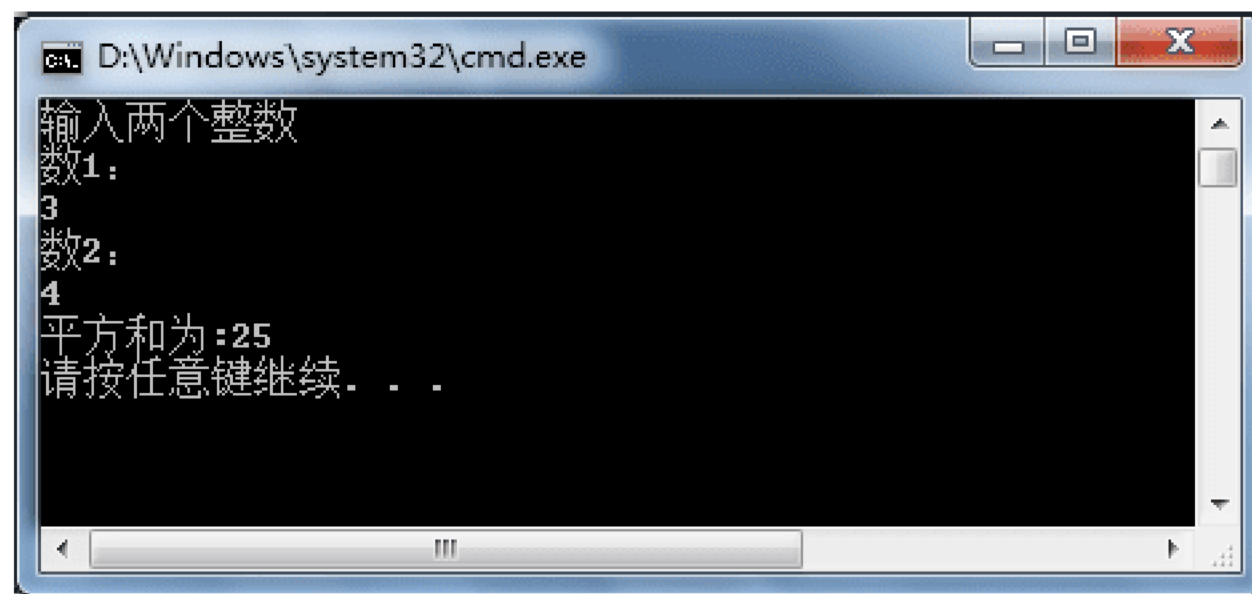


图 8.22　分步计算

8.4.3　显示数组元素

【例 8.18】 在窗体上显示一个 3 行 4 列的数组，输入要显示数组元素的所在行数和列数（该元素的坐标），将在窗体上显示该数组元素值。





代码如下：

实例位置：光盘\MR\Instance\08\8.18

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
using std::cin;
using std::cout;
using std::endl;
using std::setw;                                    //输出宽度
using std::setiosflags;                             //设置输出格式
int _tmain(int argc, _TCHAR* argv[])
{
    int a[3][4] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    int (*pt)[4], i, j;
    cout<<"The array is:"<<endl;
    for(i = 0; i < 3; i++)                          //输出数组
    {
        for(j = 0; j < 4; j++)
            cout<<setiosflags(std::ios::left)<<setw(4)<<a[i][j]<<"   ";
        cout<<endl;
    }
    cout<<"Plesase input the position like: i = ,j = \n";
    pt = a;                                         //数组指针指向数组
    cin>>i>>j;
    //根据 i，j 的值调整指针 pt 的指向，输出其指向的元素
    cout<<"a["<<i<<","<<j<<"] = "<<*(*(pt + i) + j)<<endl;
    return 0;
}
```

程序运行结果如图 8.23 所示。

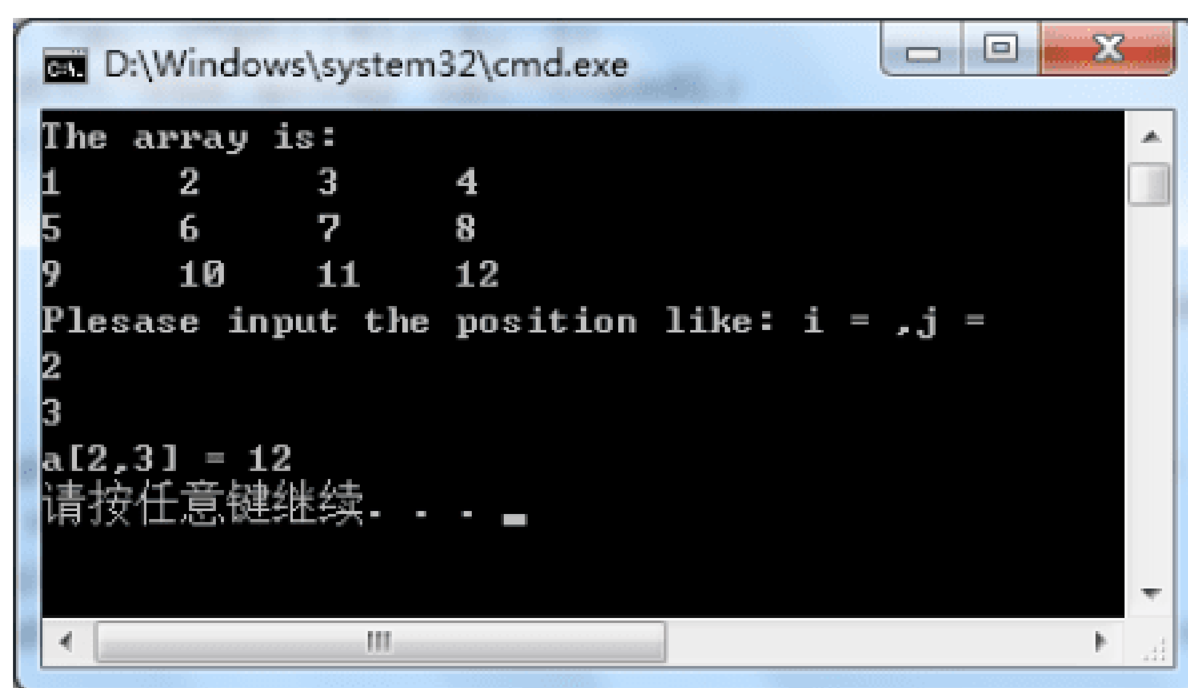


图 8.23　分步计算

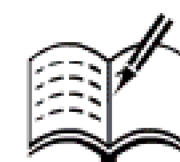
8.5　本章常见错误

8.5.1　文字常量区不可修改

下面代码中，我们希望修改指针 p 所指向的地址中的值，将“1”修改为“x”，结果运行出错。



```
char *p = "12345";                    //定义一个指针指向一个字符串
*p = 'x';                             //将 p 指向的地址赋值 x
printf("%c\n",*p);                    //输出 p 指向的地址中的字符
```



此处指针 p 所指向的地址中的值是不可更改的。因为我们并没有手动申请内存存放字符串，而是直接用指针 p 去保存字符串“12345”的首地址，此时该字符串保存在内存的文字常量区，该区内容不可被修改。

8.5.2　重复释放内存，错误提示“Debug Assertion Failed!”

如果两个指针指向同一块内存，只需要释放一次即可，然后将两个指针赋空。用第一个指针释放了这块内存之后，第二个指针指向的地址就不可用了，再用这个指针去释放就会出现重复释放。

8.5.3　释放空间以后，记得给指针赋空

如下代码，花括号里的语句总是不执行。

```
int *p = new int;                     //申请四字节内存，用 p 保存该空间首地址
delete p;                             //通过 p 释放该内存
if(p==NULL)
{…}
```

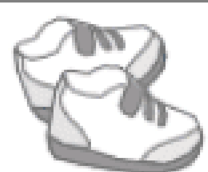
该内存被释放以后，指针 p 不是指向 NULL，而是指向一个随机地址，所以 if 语句从不执行。在释放内存空间以后，要记得给指针手动赋空。

8.5.4　(*p)--输出的不是想要的值

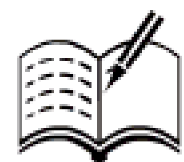
```
int a = 10;
int *p = &a;
cout<<(*p)--;                         //输出的还是 10
```

*p 是将 a 的值 10 取出，再自减，结果应该是 9，为什么输出的还是 10。这里自减运算符在变量右侧，要先用变量的值，再将该值减 1，即先输出*p 的值，再将*p 的值减 1。所以此次输出的是减 1 之前的值，如果再输出*p，结果就是 9 了。

```
cout<<(*p)--;                         //输出 10
cout<<(*p);                           //输出 9
```



8.6 本章小结



指针是 C++中的难点，它可以控制变量，可以操作数组，可以指向函数，所以一定要理解指针的基本用法。本章讲述的都是指针的基本用法，要从概念上区分指针与变量的区别。同时，还应该注意指针的正确使用方法，掌握灵活、高效、安全的内存调用的技巧。

8.7 跟我上机

参考答案：光盘\MR\跟我上机

编写一个函数，实现将字符串“123456789”逆序输出显示。实现如下：

```
#include <iostream>
#include <cstring>
#define STRING "123456789"                         //定义字符串宏
using namespace std;
void opposite(int length,char * &p)                //逆序输出函数
{
    char *pOut = new char[length];                 //申请空间保存要输出的字符串
    pOut[strlen(p)] = '\0';                        //字符串末尾加结束符
    int i = strlen(p)-1;                           //计算最后一个字符的位置
    while('\0' != *p)                              //取到\0 为止
    {
        pOut[i--] = *p++;                          //逆序存放到申请的空间中
    }
    cout<<"逆序之后：\n"<<pOut<<endl;              //输出逆序之后的字符串
}
int main()
{
    char *p = STRING;
    int length = strlen(p)+1;
    cout<<"逆序之前：\n"<<p<<endl;
    opposite(length,p);                            //调用逆序输出函数
    return 0;
}
```

第9章

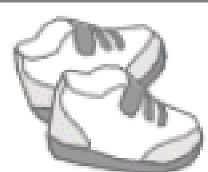
C++中的引用

（视频讲解：14分钟）

引用实际上是一种隐式指针，它为对象建立一个别名。引用是C++初学者比较容易迷惑的概念，它几乎拥有指针所有的功能，但是语法更加简单。本章学习引用，并且弄清它与指针的区别。

本章能够完成的主要范例（已掌握的在方框中打勾）

- □ 掌握引用的概念
- □ 掌握引用和指针的区别
- □ 通过引用交换数值
- □ 使用引用作为参数



9.1 引 用 概 述



Note

9.1.1 引用的概念

在 C++11 标准中提出了左值引用的概念，如果不加特殊声明，一般认为引用指的都是左值引用。

引用实际上是一种隐式指针，它为对象建立一个别名，通过操作符&来实现，引用的形式如下：

```
数据类型 & 表达式;
```

例如：

```
int a = 10;
int &ia = a;
ia = 2;
```

上面语句定义了一个引用变量 ia，它是变量 a 的别名，对 ia 的操作与对 a 的操作完全一样。ia=2 把 2 赋给 a，&ia 返回 a 的地址。执行 ia=2 和执行 a=2 等价。

使用引用的说明：

- ☑ 一个 C++引用被初始化后，无法使用它再去引用另一个对象，它不能被重新约束。
- ☑ 引用变量只是其他对象的别名，对它的操作与原来对象的操作具有相同作用。
- ☑ 指针变量与引用有两点主要区别：
  - ➢ 指针是一种数据类型，而引用不是一个数据类型，指针可以转换为它所指向变量的数据类型，以便赋值运算符两边的类型相匹配；而在使用引用时，系统要求引用和变量的数据类型必须相同，不能进行数据类型转换。
  - ➢ 指针变量和引用变量都用来指向其他变量，但指针变量使用的语法要复杂一些；而在定义了引用变量后，其使用方法与普通变量相同。

例如：

```
int a;
int *pa = &a;
int &ia = a;
```

- ☑ 引用应该初始化，否则会报错。

例如：

```
int a;
int b;
int &a;
```





Note

编译器会报出 references must be initialized 这样的错误，造成编译不能通过。

下面通过实例使读者更好地了解引用的使用，实例输出引用的功能。

【例 9.1】 输出引用。

实例位置：光盘\MR\Instance\09\9.1

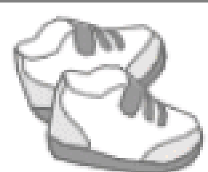
```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int a;
	int & ref_a = a;			//定义了一个 a 的引用
	a=100;				//对 a 初始化
	cout << "a="<< a <<endl;
	cout << "ref_a="<< ref_a << endl;
	a=2;
	cout << "a="<< a <<endl;
	cout << "ref_a="<< ref_a << endl;
	int b=20;
	ref_a=b;
	cout << "a="<< a <<endl;
	cout << "ref_a="<< ref_a << endl;
	ref_a--;				//对 a 的引用做自减操作
	cout << "a="<< a <<endl;
	cout << "ref_a="<< ref_a << endl;
}
```

程序声明了变量 a 和一个对变量 a 的引用 ref_a，通过不断地改变变量 a 和引用 ref_a 的值使读者了解引用的使用，然后将改变的结果输出，程序运行结果如图 9.1 所示。

图 9.1　输出引用

9.1.2　引用就是别名常量

引用就像是中国古代的女人一样，一旦嫁给某人，就要跟他一辈子，因此假如你定义了某个变量的别名，那么该别名就永远属于这个变量，它会忠心耿耿地跟随该变量，即使中间有别的变量来收买它，它也不会更换自己的主人。不过它会收下该变量的金钱，从而导致它的主人也被

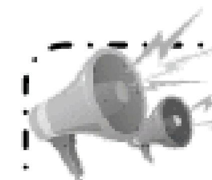


牵连。

一旦为某个变量取了别名，那么该别名将会忠心耿耿地跟随此变量，例如为变量 a 取了别名 ra，那么你无法再为变量 b 取名为 ra。别名常量指的就是这个意思，别名的主人是不能改变的，但是别名的值是可以改变的。



Note



注意：

要注意区别别名和别名的值，别名是外号，属于谁就是谁的，不可更改，别名的值是数据，数据是可以修改的。

9.1.3 右值引用

右值引用是 C++11（即 C++0x）新增加的一个非常量的引用类型。它的形式为：

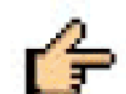
```
类型 && i = 被引用的对象;
```

左值与右值的区别在于，右值是临时变量，如函数的返回值，并且无法被改变。 例如：

```
#include "stdafx.h"
#include <iostream>
int get()
{
	int i =4;
	return i;
}
int main()
{
	int k =3;
	//int a =++(get());		//编译出错
	//int a =++(get()+k);		//编译出错
	return 0;
}
```

那么什么是右值引用呢？右值引用可以理解为右值的引用，当右值引用初始化后，临时变量消失。

【例 9.2】 右值引用的定义。

 **实例位置：光盘\MR\Instance\09\9.2**

```
#include "stdafx.h"
#include <iostream>
int get()
{
	int i = 4;
	return i;
}
int main()
```






```
{
	int &&k = get()+4;
	// int &i = get()+4;   //出错
	k++;
	std::cout<<"k 的值"<<k<<std::endl;
	return 0;
}
```

程序运行结果如图 9.2 所示。



图 9.2　执行结果

右值引用只可以初始化于右值，但右值引用实质上是一个左值，它具有临时变量的数据类型。与左值引用相同的是：

- ☑ 一个右值引用被初始化后，无法使用它再去引用另一个对象，它不能被重新约束。
- ☑ 右值引用初始化后，具有该类型数据的所有操作。

9.2　引用在函数中的应用

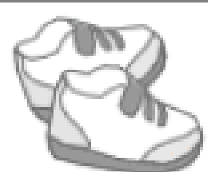
9.2.1　引用作为函数的形参

在 C++语言中，函数参数的传递方式主要有两种，分别为值传递和引用传递。所谓值传递，是指在函数调用时，将实际参数的值赋值一份传递到调用函数中，这样如果在调用函数中修改了参数的值，其改变不会影响到实际参数的值。而引用传递则恰恰相反，如果函数按引用方式传递，在调用函数中修改了参数的值，其改变会影响到实际参数。

【例 9.3】 通过引用交换数值。

实例位置：光盘\MR\Instance\09\9.3

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void swap(int & a,int & b)
{
	int tmp;
	tmp=a;
	a=b;
	b=tmp;
}
void main()
{
	int x,y;
	cout << "请输入 x" << endl;
```





Note

```
        cin >> x;
        cout << "请输入 y" << endl;
        cin >> y;
        cout<<"通过引用交换 x 和 y"<<endl;
        swap(x,y);
        cout << "x=" << x <<endl;
        cout << "y=" << y <<endl;
}
```

程序运行结果如图 9.3 所示。



图 9.3　通过引用交换数值

程序中自定义函数 swap，函数定义了两个引用参数，用户输入两个值，如果第一次输入的数值比第二次输入的数值小，则调用 swap 函数交换用户输入的数值。如果使用值传递方式，swap 函数就不能实现交换。

9.2.2　指针与引用

正如所见，引用传递参数与指针传递参数达到同样的目的。指针传递参数也属于一种值传递，传递的是指针变量的副本。如果使用指针的引用，就可达到在函数体内改变指针地址的目的。

【例 9.4】 指针的引用作参数。

实例位置：光盘\MR\Instance\09\9.4

```
#include "stdafx.h"
#include <iostream>
using std::cout;
using std::endl;
static int global=16;              //静态全局变量
void getMax(int* &p)
{
        if(*p<global)
        {
                delete p;          //释放内存
                p = &global;       //相当于 pl1 的引用改变了
        }
}
void getMin(int *p)
```






```
{
    if(*p>global)
    {
        delete p;           //释放了 pI2 所指向的内存
        p = &global;        //副本值改变了，pI2 无变化
    }
}
int main()
{
    int* pI1 = new int;
    int* pI2 = new int;
    cout<<"pI1 指向的地址:"<<pI1<<endl;
    cout<<"pI2 指向的地址:"<<pI2<<endl;
    *pI1 = 15;              //global 较大
    *pI2 = 18;              //global 较小
    cout<<"全局变量 global 的地址:"<<&global<<endl;
    cout<<"将 pI1 与 pI2 分别带入 getMax 与 getMin 函数"<<endl;
    getMax(pI1);
    getMin(pI2);
    cout<<"pI1 指向的地址:"<<pI1<<endl;
    cout<<"pI2 指向的地址:"<<pI2<<endl;
    cout<<"*pI1 的值:"<<*pI1<<endl;
    cout<<"*pI2 的值:"<<*pI2<<endl;
    return 0;
}
```

程序运行结果如图 9.4 所示。



图 9.4　执行结果

getMax 函数通过传递指针的引用改变了指针的地址，指针 pI1 的地址最终指向了全局变量。而通过按值传递指针的 getMin 函数中，只能够改变内存的内容，对内存执行操作，并不能改变指针所指向的地址。

引用类型不存在指针，例如：

```
int& *p
```

上面的声明是非法的，从左向右延伸的意思为这是指向 int 型数据别名的指针类型变量。别名无法被指针指向，所以是非法的。指针可以指向变量的引用，相当于指针指向了该变量。





Note

9.2.3　右值引用传递参数

使用字面值，如 1、3.15f、true 或者表达式等临时变量作为函数实参传递时，按左值引用传递参数都会被编译器阻止。而进行值传递时，将产生一个和参数同等大小的副本。C++11 提供右值引用传递参数，不要申请局部变量，也不会产生参数副本。

【例 9.5】 右值引用传递参数。

实例位置：光盘\MR\Instance\09\9.5

```
#include "stdafx.h"
#include <iostream>
using namespace std;
static float  global = 1.111f;          //静态全局变量
void offset(float && f)                 //右值引用
{
	global += f;
}
float getFloat()
{
	float f = 4.444f;
	return f;
}
void offset(float& f)                   //重载了 offset 函数（左值引用）
{
	global -= f;
}
int main()
{
	float u = 10.000f;
	cout<<"global:"<<global<<endl;
	offset(3.333f);                     //调用右值引用函数
	cout<<"global:"<<global<<endl;
	offset(getFloat()+2.222);
	cout<<"global:"<<global<<endl;
	offset(u);                          //左值引用
	cout<<"global:"<<global<<endl;
	return 0;
}
```

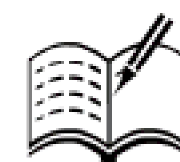
程序运行结果如图 9.5 所示。

程序中重载了 offset 函数，可以看到此函数的功能是通过函数的参数改变全局变量的值。右值引用只接收右值实参，可以将它看作是临时变量的别名，不会将临时变量再复制 1 次，比按值传递提高了效率。



图 9.5　执行结果





9.3　综合应用

【例 9.6】 本例实现输入 3 个整数，将这 3 个整数按照由大到小的顺序输出，显示在屏幕上。程序设计步骤如下：

（1）创建控制台应用程序。

（2）创建自定义函数 swap，用于实现数据的交换。代码如下：

实例位置：光盘\MR\Instance\09\9.6

```
void swap(int &p1,int &p2)
{
	int temp;
	temp = p1;
	p1 = p2;
	p2 = temp;
}
```

（3）创建自定义函数 exchange，用于实现比较数值大小，并调用自定义函数 swap，交换数据的位置。代码如下：

```
void exchange(int &pt1,int &pt2,int &pt3)
{
	if(pt1<pt2)
		swap(pt1,pt2);
	if(pt1<pt3)
		swap(pt1,pt3);
	if(pt2<pt3)
		swap(pt2,pt3);
}
```

（4）创建 main 函数作为程序的入口函数，并在该函数中调用 exchange 函数，实现对输入的 3 个数据比较大小并交换位置。代码如下：

```
void main()
{
	int a,b,c;
	puts("Please input three key numbers you want to rank:");
	cin>>a>>b>>c;
	exchange(a,b,c);
	cout<<a<<" "<<b<<" "<<c;
}
```

程序运行结果如图 9.6 所示。





Note

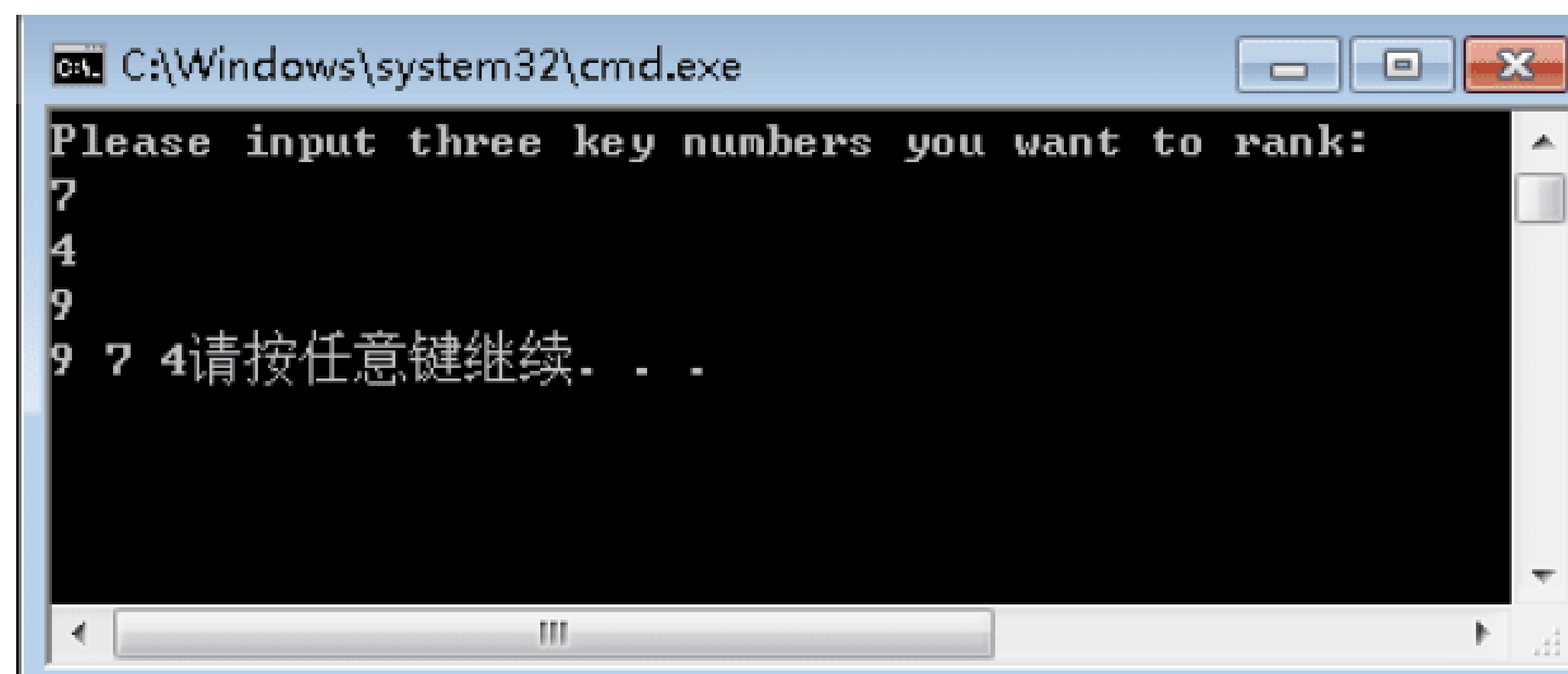


图 9.6　使用左值引用实现整数排序

9.4　本章常见错误

9.4.1　指针和引用分别应该什么时候用

指针和引用分别应该在什么时候用呢？

一个指针变量可以指向 NULL，表示它不指向任何变量地址，但是引用必须在声明时就和一个已经存在的变量相绑定，而且这种绑定不可改变。

9.4.2　在哪里创建，就在哪里释放指针

有时可能将指针删除两次或者忘记删除指针。为了避免指针混淆，应“在哪里创建，就在哪里释放”指针。因此，在 main 函数中创建一个堆中对象，然后按引用的方式传递到 func()函数中，在 func()函数中对该对象操作完毕后返回该对象，然后在 main 函数中释放该对象。这样就实现了在 main 函数中创建，在 main 函数中释放。

9.4.3　指针和引用混合使用

例如：

```
int *&r = new int;
```

也可以在函数的参数中混合使用，例如：

```
int *func(int &one,int *two,int x);
```

该行的 func 函数有 3 个参数，第一个是 int 型变量的别名 one，第二个是指向 int 型变量的指针 two，第三个是整型参数 x。func 函数的返回值是一个执行 int 型变量的指针。



9.4.4　指针的特殊写法



```
int*r,ra;
```

初学者也许会认为 r 和 ra 都是指向 int 型变量的指针，假如这么想的话，就错了，其实只有 r 是指向 int 型变量的指针，而 ra 是 int 型变量。因此最好将*r 与 t 分开写，例如：

```
int *r, ra;
```

这样就不会产生歧义了。

9.5　本 章 小 结

本章详细介绍了引用的分类和函数参数传递等知识。通过学习引用传递参数，更深入地理解函数运行过程。同时，还应该注意引用与指针的区别。

9.6　跟 我 上 机

参考答案：光盘\MR\跟我上机

通过引用交换数值。实现代码如下：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void swap(int & a,int & b)
{
	int temp;
	temp = a;
	a = b;
	b = temp;
}
void main()
{
	int x,y;
	cout<<"input two number:"<<endl;
	cin>>x;
	cin>>y;
	if(x<y)
		swap(x,y);
	cout<<"x="<<x<<endl;
	cout<<"y="<<y<<endl;
}
```

第10章

使用数组获取连续空间

（视频讲解：56分钟）

数组用来储存多个相同数据类型数据，它是这些数据的集合体。而使用数组实质上就是使用数组中的每个元素。它提供的顺序储存结构使原本毫无关联的变量联系起来，是算法执行的重要工具。

本章能够完成的主要范例（已掌握的在方框中打勾）

- ☐ 数组的初始化
- ☐ 将二维数组的行列交换
- ☐ 使用 strcpy 函数复制字符串
- ☐ 使用 strcmp 函数比较两个字符串大小
- ☐ 使用 gets 和 puts 输入/输出字符串
- ☐ 动态获得斐波纳契数列
- ☐ 修改 string 字符串的单个字符
- ☐ 比较两个 string 字符串大小





10.1　一 维 数 组

10.1.1　声明一维数组

在程序设计中，将同一数据类型的数据按一定形式有序地组织起来，这些有序数据的集合就称为数组。一个数组有一个统一的数组名，可以通过数组名和下标来唯一确定数组中的元素。

一维数组的声明形式如下：

```
数据类型  数组名[常量表达式]
```

例如：

```
int a[10];              //声明一个整型数组，有 10 个元素
char name[128];         //声明一个字符数组，有 128 个元素
float price[20];        //声明一个浮点数组，有 20 个元素
```

使用数组的说明：

☑　数组名的定名规则和变量名相同。

☑　数组名后面的括号是方括号，方括号内是常量表达式。

☑　常量表达式表示元素的个数，即数组的长度。

☑　定义数组的常量表达式不能是变量，因为数组的大小不能动态定义。

```
int a[i];               //不合法
```

10.1.2　一维数组的元素

数组元素的一般形式如下：

```
数组名[下标]
```

例如：

```
int a[10];              //声明数组
```

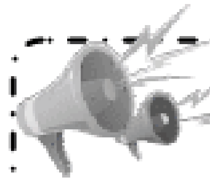
a[0]、a[1]、a[2]、a[3]、a[4]、a[5]、a[6]、a[7]、a[8]、a[9]是对数组 a 中 10 个元素的引用。

一维数组元素的说明：

☑　数组元素的下标起始值为 0 而不是 1。

☑　a[10]是不存在的数组元素，使用 a[10]非法。





注意：

a[10]属于下标越界，下标越界容易造成程序瘫痪。



Note

10.1.3 初始化一维数组

数组元素初始化的方式有两种，一种是对单个元素逐一赋值，另一种是使用聚合方式赋值。

1．单一数组元素赋值

a[0]=0 就是对单一数组元素赋值，也可以通过变量控制下标的方式进行赋值，例如：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	char a[3];			//定义一个字符型数组
	a[0]='a';			//给第 0 个元素赋字符 a
	a[2]='c';			//给第 2 个元素赋字符 c
	int i=0;
	cout << a[i] << endl;
}
```

程序运行结果如图 10.1 所示。

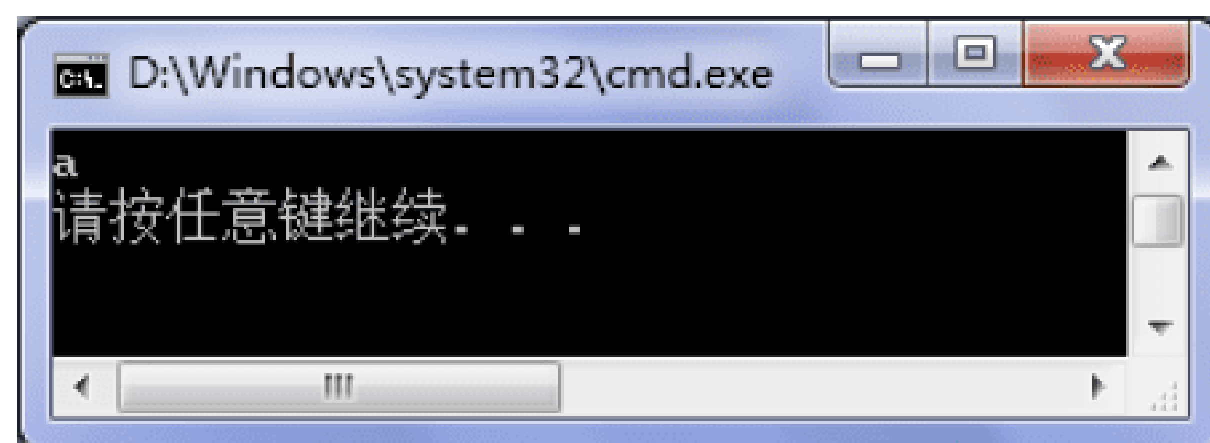


图 10.1　单一数组元素赋值

2．聚合方法赋值

数组不仅可以逐一对数组元素赋值，还可以通过大括号进行多个元素的赋值。例如：

```
int a[12]={1,2,3,4,5,6,7,,8,9,10,11,12};
```

或

```
int a[]={1,2,3,4,5,6,7,,8,9,10,11,12};		//编译器能够获得数组元素个数
```

或

```
int a[12]={1,2,3,4,5,6,7};			//前 7 个元素被赋值，后面 5 个元素的值默认用 0 填充
```

下面通过实例来看一下如何为一维数组的数组元素赋值。





Note

【例 10.1】 一维数组赋值。

实例位置：光盘\MR\Instance\10\10.1

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int i,a[10];
	for(i=0;i<10;i++)			//利用循环，分别为 10 个元素赋值
		a[i]=i;
	for(i=0;i<10;i++)			//将数组中的 10 个元素输出到显示设备
		cout << a[i] << endl;
}
```

程序运行结果如图 10.2 所示。

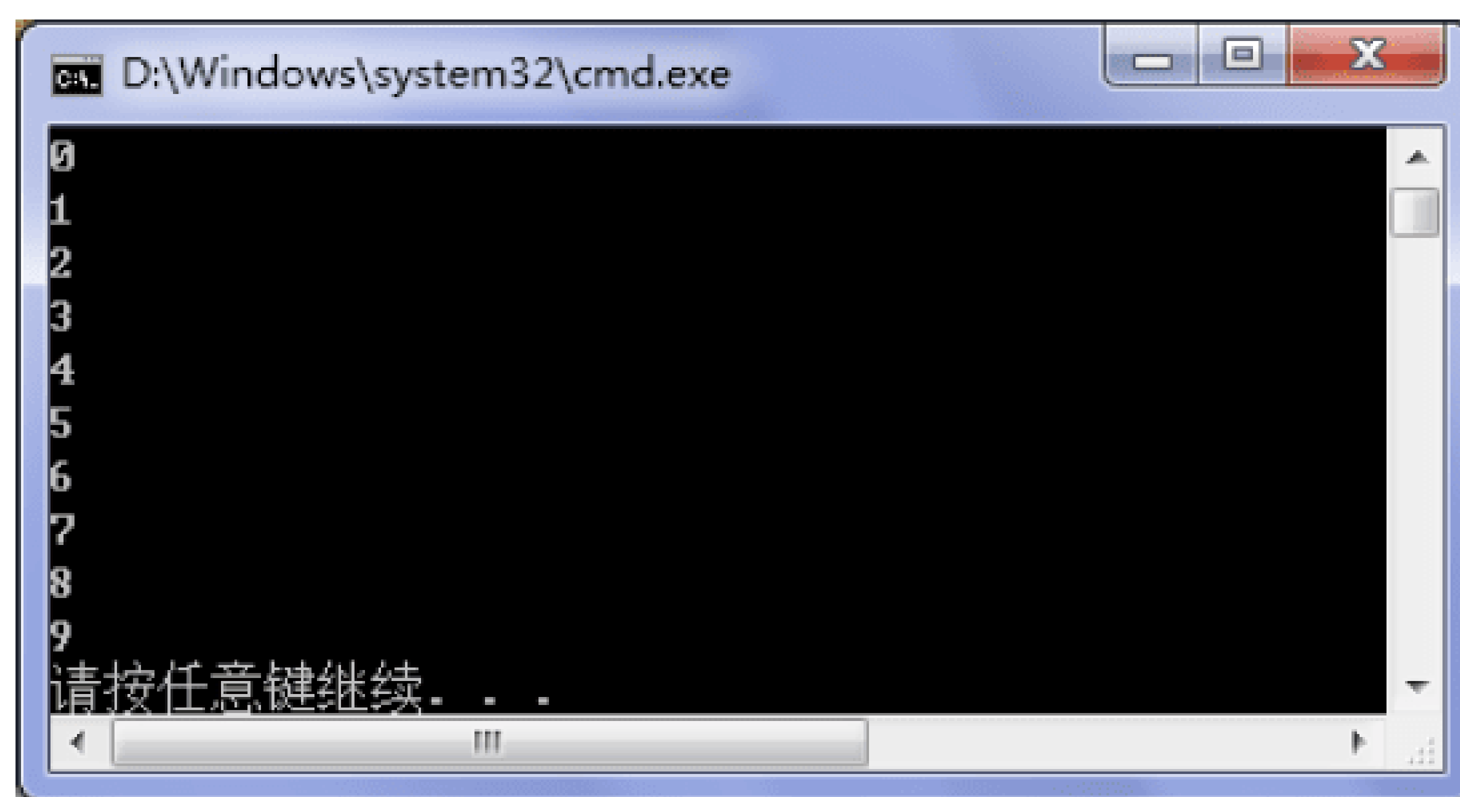


图 10.2　一维数组赋值

程序实现通过 for 循环将 int a[10]定义的数组中的每个元素赋值，然后再循环通过 cout 函数将数组中的元素值输出到显示设备。

10.2　二 维 数 组

10.2.1　声明二维数组

二维数组声明的一般形式为：

```
数据类型 数组名[常量表达式 1][常量表达式 2]
```

例如：

```
int a[3][4];				//声明具有 3 行 4 列元素的整型数组
float myArray[4][5];			//声明具有 4 行 5 列元素的浮点数组
```





一个一维数组描述的是一个线性序列，二维数组则描述的是一个矩阵。常量表达式 1 代表行的数量，常量表达式 2 代表列的数量。

二维数组可以看作是一种特殊的一维数组，如图 10.3 所示，虚线左侧为 3 个一维数组的首元素，二维数组是由 A[0]、A[1]、A[2]这 3 个一维数组组成，每个一维数组都包含 4 个元素。

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[2][0]	A[2][1]	A[2][2]	A[2][3]

图 10.3　二维数组

使用数组的说明：

☑　数组名的定名规则和变量名相同。

☑　二维数组有两个下标，所以要有两个中括号。

```
int a[3,4]                    //不合法
int a[3:4]                    //不合法
```

☑　下标运算符中的整数表达式代表数组每一个维的长度，它们必须是正整数，其乘权确定了整个数组的长度。

例如：

```
int a[3][4]
```

其长度就是 3*4=12。

☑　定义数组的常量表达式不能是变量，因为数组的大小不能动态定义。

```
int a[i][j];                  //不合法
```

10.2.2　引用二维数组元素

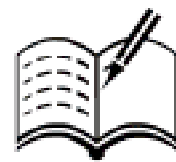
二维数组元素形式为：

```
数组名[下标][下标]
```

二维数组元素和一维数组基本相同。例如：

```
a[2-1][2*2-1]                 //合法
a[2,3],a[2-1,2*2-1]           //不合法
```





10.2.3　初始化二维数组

二维数组元素初始化的方式和一维数组相同，也分为单个元素逐一的赋值和使用聚合方式赋值。

例如：

```
myArray[0][1]=12;                            //单个元素初始化
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};    //使用聚合方式赋值
```

使用聚合方式给数组赋值等同于分别对数组中的每个元素进行赋值。例如：

```
int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
```

等同于执行如下语句：

```
a[0][0]=1;a[0][1]=2;a[0][2]=3;a[0][3]=4;
a[1][0]=5;a[1][1]=6;a[1][2]=7;a[1][3]=8;
a[2][0]=9;a[2][1]=10;a[2][2]=11;a[2][3]=12;
```

二维数组中元素排列的顺序是按行存放，即在内存中先顺序存放第一行的元素，再存放第二行的元素。例如，“int a[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};”的赋值顺序是：先给第一行元素赋值 a[0][0]->a[0][1]->a[0][2]->a[0][3]，再给第二行元素赋值 a[1][0]->a[1][1]->a[1][2]->a[1][3]，最后给第三行元素赋值 a[2][0]->a[2][1]->a[2][2]->a[2][3]。数组元素的位置以及对应数值如图 10.4 所示。

A[0][0]	A[0][1]	A[0][2]	A[0][3]
A[1][0]	A[1][1]	A[1][2]	A[1][3]
A[2][0]	A[2][1]	A[2][2]	A[2][3]

数组位置

1	2	3	4
5	6	7	8
9	10	11	12

数值位置

图 10.4　数组位置对应的数值

使用聚合方式赋值，还可以按行进行赋值。例如：

```
int a[3][4]={{1,2,3,4},{5,6,7,8},{9,10,11,12}};    //整体全部赋值
```

二维数组可以只对前几个元素赋值。例如：

```
a[3][4]={1,2,3,4};                           //相当于给第一行赋值，其余数组元素全为 0
```

数组元素是左值，可以出现在表达式中，也可以对数组元素进行计算。例如：



```
b[1][2]=a[2][3]/2;                                       //对数组元素进行计算和赋值
```

下面通过实例来熟悉一下二维数组的操作，实例将实现将二维数组中行数据和列数据相互置换的功能。



Note

【例 10.2】 将二维数组行列对换。

实例位置：光盘\MR\Instance\10\10.2

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int fun(int array[3][3])                                 //数组作参数，被优化成指针类型
{
        int i,j,t;
        for(i=0;i<3;i++)                                 //控制行
                for(j=0;j<i;j++)                         //控制列
                {
                        t=array[i][j];                   //交换行列坐标
                        array[i][j]=array[j][i];
                        array[j][i]=t;
                }
                return 0;
}
void main()
{
        int i,j;
        int array[3][3]={{1,2,3},{4,5,6},{7,8,9}};       //定义 3 行 3 列的数组
        cout << "Converted Front" <<endl;
        for(i=0;i<3;i++)                                 //循环遍历数组元素
        {
                for(j=0;j<3;j++)
                        cout << setw(7) << array[i][j] ;  //按 7 个宽度输出
                cout<< endl;
        }
        fun(array);                                      //调 fun 函数，交换行列
        cout << "Converted result" <<endl;
        for(i=0;i<3;i++)
        {
                for(j=0;j<3;j++)
                        cout << setw(7) << array[i][j] ;  //输出交换之后的数组
                cout<< endl;
        }
}
```

程序运行结果如图 10.5 所示。

程序首先输出二维数组 array 中的元素，然后调用自定义函数 fun 将数组中的行元素转换为列元素，最后输出转换后的结果。





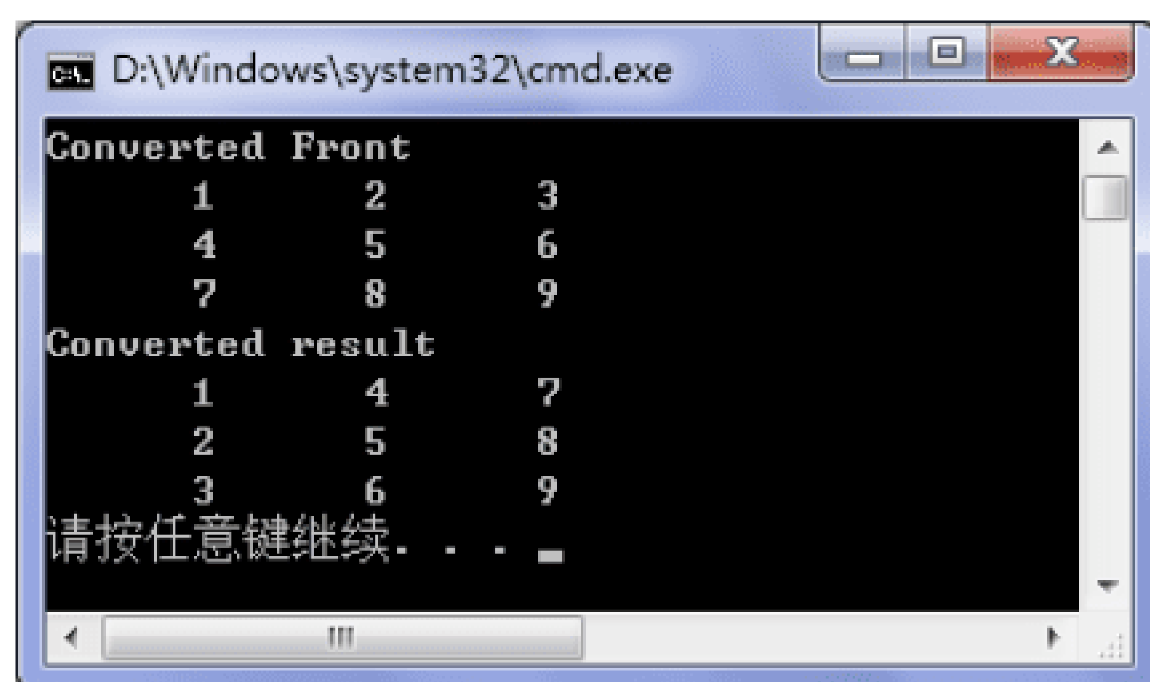


图 10.5　将二维数组行列对换

10.3　字符数组

用来存放字符数据的数组是字符数组，字符数组中的一个元素存放一个字符。字符数组具有数组的共同属性。由于字符串应用广泛，C 和 C++专门为它提供了许多方便的用法和函数。

10.3.1　声明一个字符串数组

```
char pWord[11];
```

表示的是容纳 11 个字符的数组。

10.3.2　字符串数组赋值

数组元素逐一赋值。例如：

```
pWord[0]='H' pWord[1]='E' pWord[2]='L' pWord[3]='L'
pWord[4]='O' pWord[5]=' ' pWord[6]='W' pWord[7]='O'
pWord[8]='R' pWord[9]='L' pWord[10]='D'
```

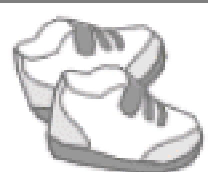
使用聚合方式赋值。例如：

```
char pWord[]={'H','E','L','L','O',' ','W','O','R','L','D'};
```

如果花括号中提供的初值个数大于数组长度，则按语法错误处理。如果初值个数小于数组长度，则只将这些字符赋给数组中前面那些元素，其余的元素自动定义为空字符。如果提供的初值个数与预定的数组长度相同，在定义时可以省略数组长度，系统会自动根据初值个数确定数组长度。

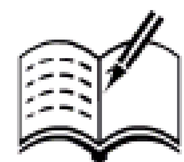
10.3.3　字符数组的一些说明

聚合方式只能在数组声明时使用。例如：



```
char pWord[5];
pWord = {'H','E','L','L','O'};                //错误（可以用 strcpy 函数复制）
```

字符数组不能给字符数组赋值。例如：



Note

```
char a[5] = {'H','E','L','L','O'};
char b[5];
a=b;                                          //错误
a[0]=b[0];                                    //正确
```

10.3.4 越界引用

字符数组常作字符串使用，作为字符串要有字符串结束符“\0”，否则，引用该字符串时会出现越界非法引用的现象。

可以使用字符串为字符数组赋值。例如：

```
char a[]= "HELLO WORLD";
```

等同于：

```
char a[]= "HELLO WORLD\0";
```

字符串结束符“\0”主要告知字符串处理函数字符串已经结束了，不需要再输出了。

下面通过实例来看一下使用字符串结束符“\0”和不使用字符串结束符“\0”的区别。

【例 10.3】 使用字符串结束符“\0”标识一个字符串的结束，防止出现非法字符。

实例位置：光盘\MR\Instance\10\10.3

未使用字符串结束符“\0”的程序，代码如下：

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
{
    int i;                                    //定义一个整型变量
    char array[12];                           //字符数组
    array[0]='a';                             //赋字符 a
    array[1]='b';                             //赋字符 b
    printf("%s\n",array);                     //输出数组元素（访问后面未初始化的内存，输出垃圾值）
}
```

程序运行结果如图 10.6 所示。

使用字符串结束符“\0”的程序，代码如下：

```
#include "stdafx.h"
#include<iostream>
```






```
using namespace std;
void main()
{
	int i;
	char array[12];
	array[0]='a';
	array[1]='b';
	array[2]='\0';                    //赋结束符，指针取值到此结束，不继续向下访问
	printf("%s\n",array);
}
```

程序运行结果如图 10.7 所示。

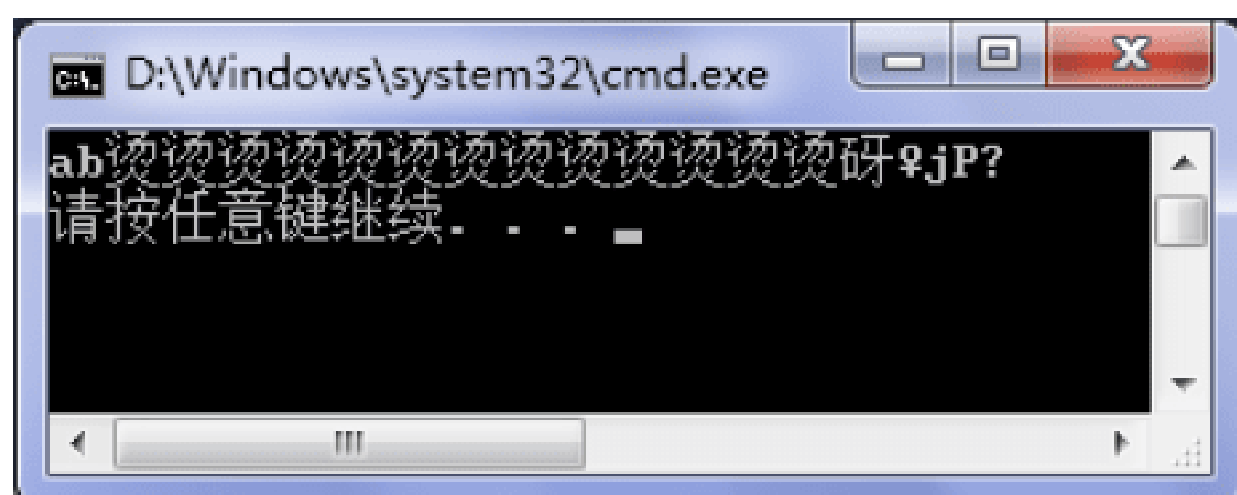


图 10.6　未使用字符串结束符“\0”

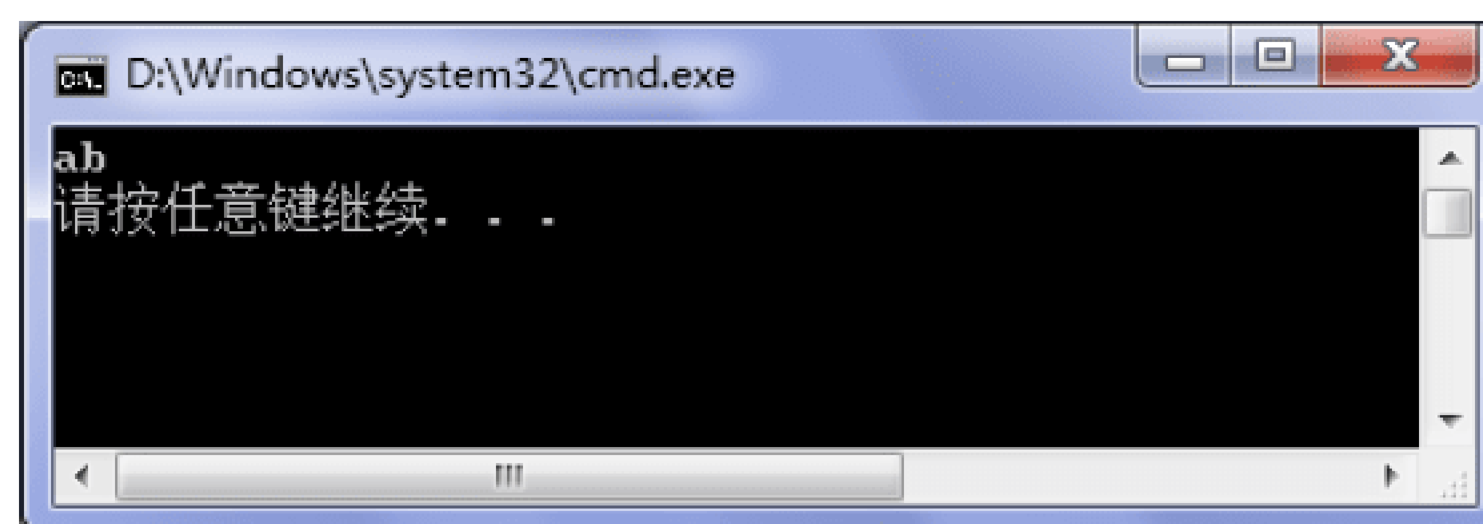


图 10.7　使用字符串结束符'\0'

printf 函数使用%s 格式可以输出字符串，如果字符串中没有结束符，函数会按整个字符数组输出。array 字符数组中只有前两个字符初始化了，所以未使用字符串结束符“\0”的程序会出现乱码。

下面通过实例来熟悉一下在程序中对字符数组的操作。

【例 10.4】 输出字符数组中内容。

实例位置：光盘\MR\Instance\10\10.4

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
{
	int i;
	char array[12]={'H','E','L','L','O',' ','W','O','R','L','D'};
	for(i=0;i<12;i++)                 //循环遍历每一个元素
		cout<<array[i];
	cout << endl;
}
```

程序运行结果如图 10.8 所示。

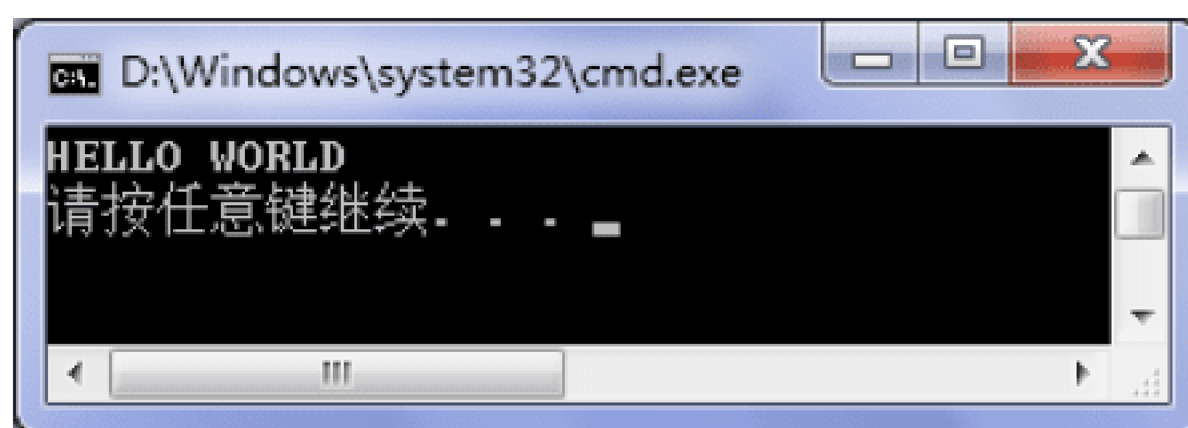
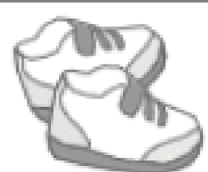


图 10.8　字符数组中内容





Note

10.3.5 字符串处理函数

☑ strlen 函数

获取字符串长度函数的格式如下：

```
strlen(字符数组名)
```

此函数的返回值是结束符“\0”前字符串长度。下面输入一个字符串，然后输出它的有效长度。程序如下：

```
#include "stdafx.h"
#include<iostream>
using std::cout;
using std::endl;
using std::cin;
void main()
{
	char str1[30],str2[20];                    //定义两个字符数组
	cout<<"请输入数组:"<< endl;
	cin>>str1;                                 //给 str1 赋值
	cout<<"字符串长度"<<strlen(str1)<<endl;    //计算 str1 数组内字符的个数
}
```

程序运行结果如图 10.9 所示。

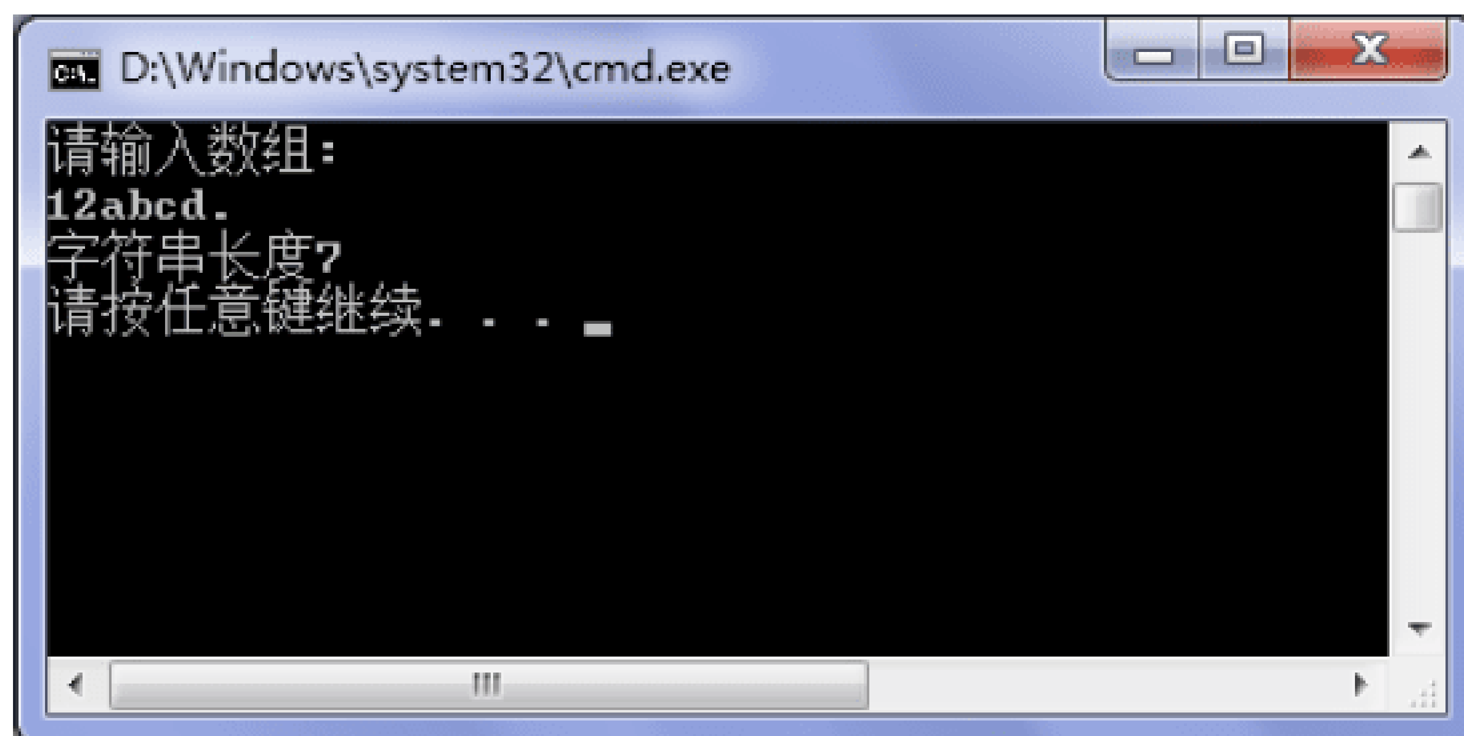


图 10.9　获取字符串长度

☑ strcat 函数

字符串连接函数 strcat 的格式如下：

```
strcat(字符数组名 1，字符数组名 2)
```

把字符数组 2 中的字符串连接到字符数组 1 中字符串的后面，并删去字符串 1 后的串结束标志“\0”。

下面通过实例使用 strcat 函数将两个字符串连接在一起。

【例 10.5】 字符串连接函数 strcat。





实例位置：光盘\MR\Instance\10\10.5

```
#include "stdafx.h"
#include<iostream>
using std::cout;
using std::endl;
using std::cin;
void main()
{
	char str1[30],str2[20];
	cout<<"请输入数组 1:"<< endl;
	cin>>str1;                          //给 str1 赋值
	cout<<"请输入数组 2"<<endl;
	cin>>str2;                          //给 str2 赋值
	if(30>strlen(str1)+strlen(str2))    //判断两个字符长度之和是否超过 30
	{
		strcat(str1,str2);              //连接 str1 和 str2
		cout <<"Now the string1 is:"<<str1<<endl;
	}
	else
		cout<<"操作失败"<<endl;
}
```

程序运行结果如图 10.10 所示。

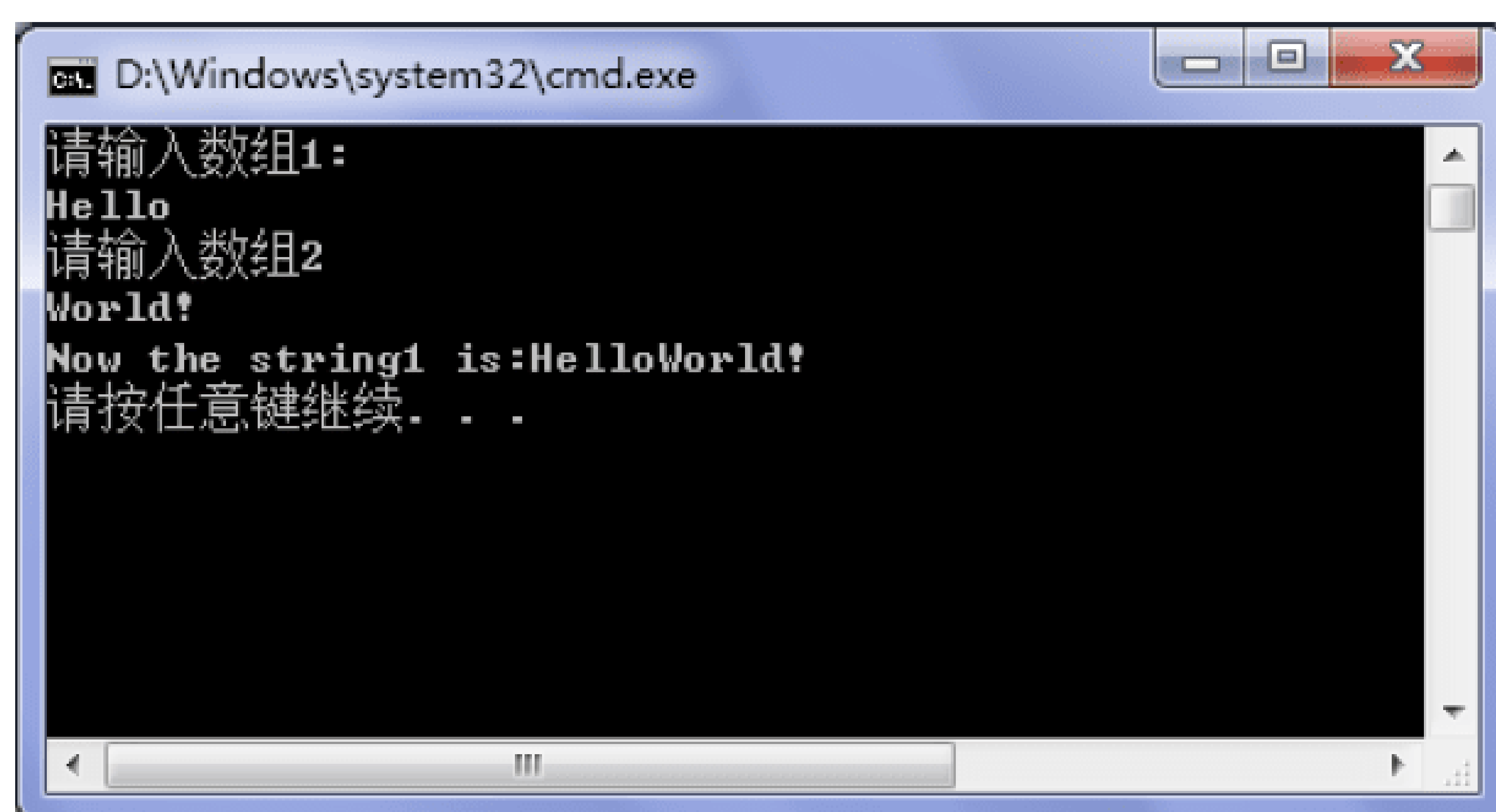


图 10.10　连接字符串



注意：

在使用 strcat 函数时要注意，字符数组 1 的长度要足够大，否则不能装下连接后的字符串。

☑　strcpy 函数

字符串复制函数 strcpy 的格式如下：

```
strcpy(字符数组名，字符串)
```

把字符串中的字符串复制到字符数组中。字符串结束标志“\0”也一同复制。

字符数组 1 应有足够的长度，否则不能全部装入所复制的字符串。



下面通过实例使用 strcpy 函数来实现字符串复制的功能。

【例 10.6】 字符串复制函数 strcpy。

实例位置：光盘\MR\Instance\10\10.6



```
#include"stdafx.h"
#include<iostream>
using namespace std;
void main()
{
	char str1[30],str2[20] = {'n','o','n','e','\0'};
	cout<<"请输入数组 1:"<< endl;
	scanf("%s",str1);	//给 str1 赋值
	strcpy(str1,str2);	//把 str2 的值复制到 str1 中
	cout<<"数组 1 的内容:"<<endl;
	printf("%s",str1);	//输出 str1 的值
}
```

程序运行结果如图 10.11 所示。无论在数组 1 中输入什么内容，执行 strcpy 都会被数组 2 中的内容代替。

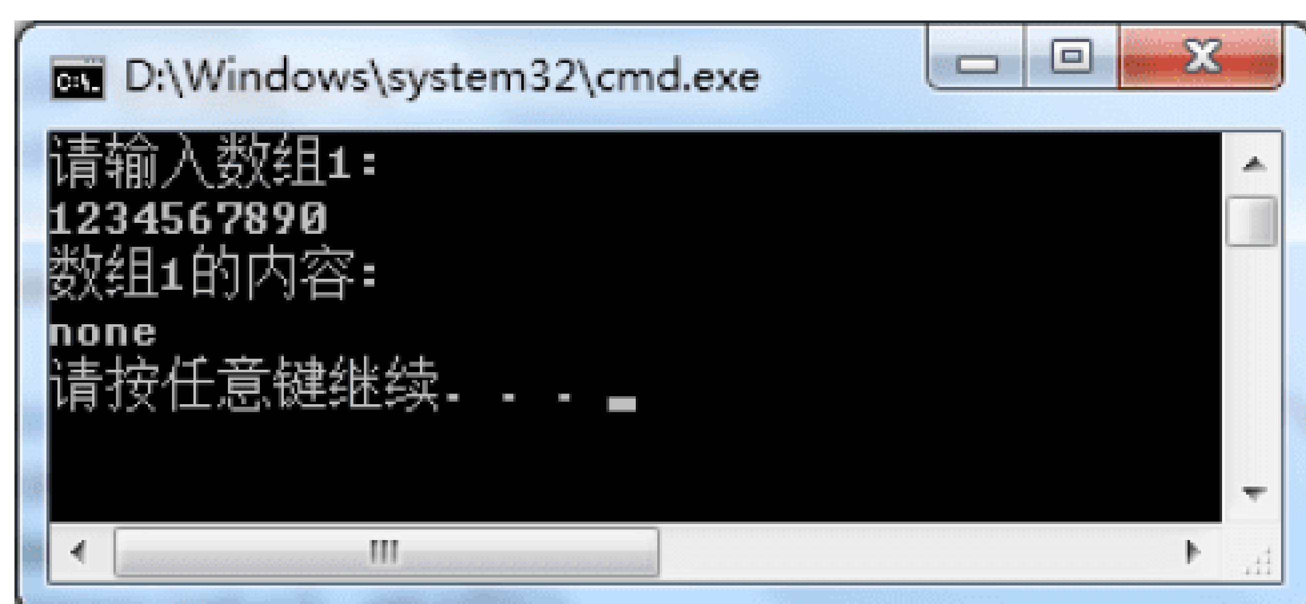


图 10.11　字符串复制



注意：

使用 strcpy 时，也可以将常量字符串作为第二个参数，赋值给第一个参数的数组。

☑ strcmp 函数

字符串比较函数 strcmp 的格式如下：

```
strcmp(字符数组名 1，字符数组名 2)
```

按照 ASCII 码，按顺序比较两个数组中的字符，并由函数返回值返回比较结果。以下是执行过程：

（1）各自选中自身的第一个字符：字符 1，字符 2。

（2）字符 1>字符 2，返回值为一正数。

（3）字符 1<字符 2，返回值为一负数。

（4）字符 1=字符 2，继续比较后面的元素，若完全相等，返回值为 0。

可用于比较两个字符串常量，或比较数组和字符串常量。例如：






```
strcmp(str1,str2);
```

该语句是两个数组进行比较：

```
strcmp(str1,"hello");
```

该语句是一个数组与一个字符串进行比较：

```
strcmp("hello","how");
```

该语句是两个字符串进行比较。

下面通过实例来看一下如何使用 strcmp 函数对字符串进行比较。

【例 10.7】 字符串比较函数 strcmp。

实例位置：光盘\MR\Instance\10\10.7

```
#include "stdafx.h"
#include<iostream>
using namespace std;
#include<string>
void main()
{
	char str1[30],str2[20];
	int i=0;
	cout<<"请输入字符串 1:"<< endl;
	gets(str1);
	cout<<"请输入字符串 2:"<<endl;
	gets(str2);
	i=strcmp(str1,str2);                //比较两个字符串的大小（ASCII 码值的大小）
	if(i>0)
	cout <<"str1>str2"<<endl;           //返回正数，输出 str1>str2
	else
	if(i<0)                             //返回负数，输出 str1<str2
	cout <<"str1<str2"<<endl;
	else
	cout <<"str1=str2"<<endl;           //返回 0，相等
}
```

程序运行结果如图 10.12 所示。

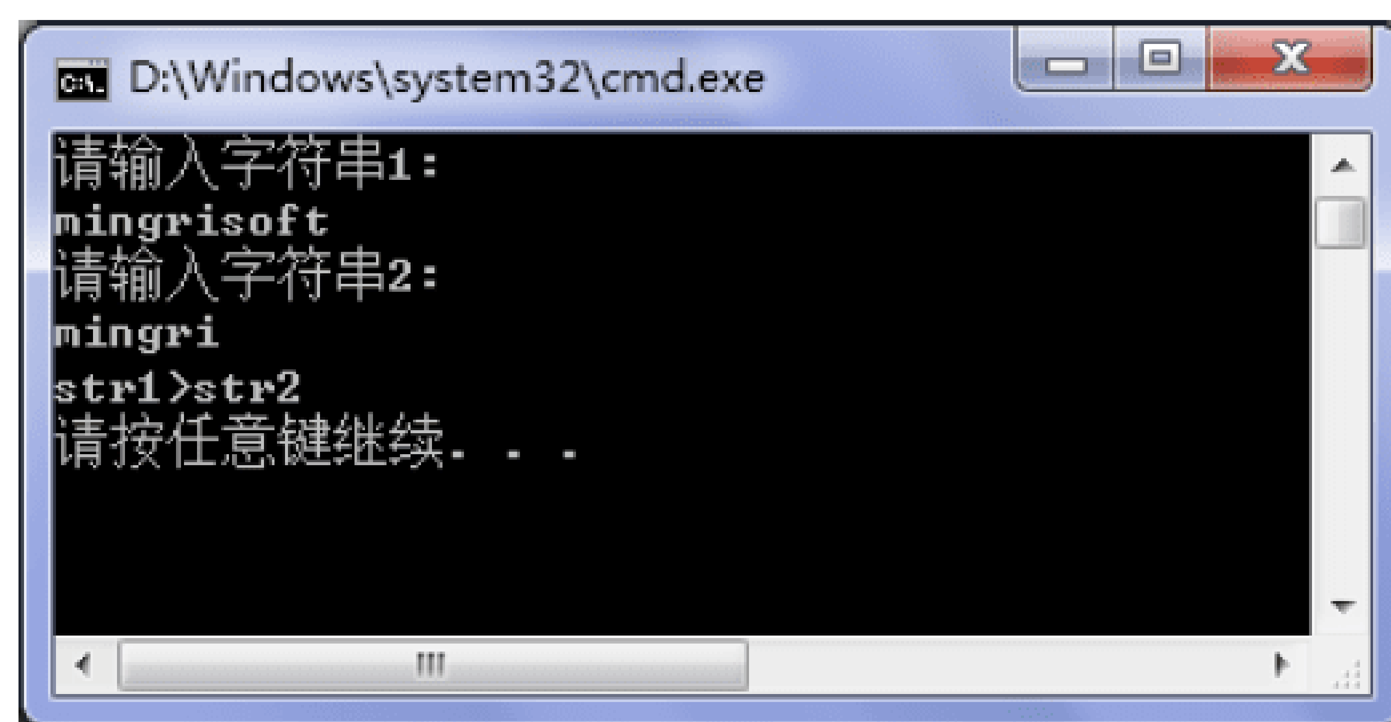


图 10.12　字符串比较





Note

☑ gets 与 puts 函数

使用标准输出函数（cin）和格式化输出函数（scanf）时都存在着这样一个问题，当输入空格时，输入的对象不会接收空格符之后的内容。

输入函数 gets 与输出函数 puts 都只以结束符“\0”为输入\输出结束的标志。下面举例来说明。

【例 10.8】 比较输入函数 gets 与 puts。

实例位置：光盘\MR\Instance\10\10.8

```
#include"stdafx.h"
#include<iostream>
using namespace std;
void main()
{
	char str1[30],str2[30],str3[30],temp[30];
	cout<<"请使用 scanf 和 cin 输入 Hello World!!"<< endl;
	scanf("%s",str1);
	cin>>str2;
	cout<<"str1:";
	printf("%s\n",str1);
	cout<<"str2:";
	cout<<str2<<endl;
	cout<<"输入流中残留了 cin 留下的空格符'，使用 gets 接收它:"<<endl;
	gets(temp);                    //给 temp 赋值
	cout<<"temp:"<<temp<<endl;
	cout<<"请使用 gets 输入 Hello World!!:"<<endl;
	gets(str3);                    //给 str3 赋值
	cout<<"str3:";
	puts(str3);                    //输出 str3
}
```

程序运行结果如图 10.13 所示。

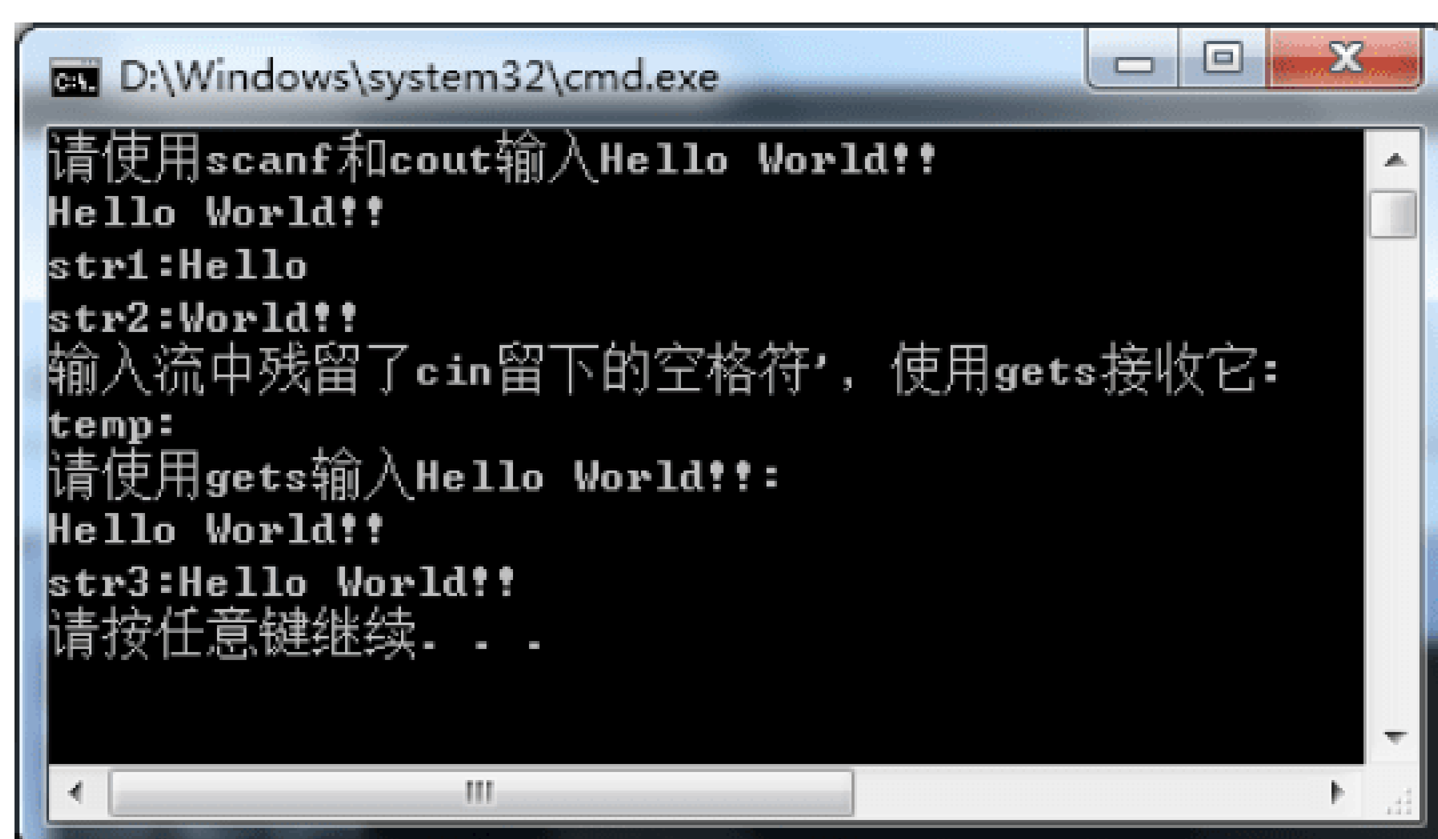
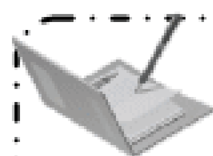


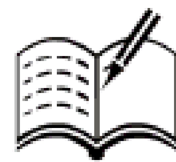
图 10.13　执行结果



说明：

标准输入流在操作完成后，流的内容仍然会保留下来。这样就每通过输入，令“World”赋值给了 str2。因为 gets 接收空格符，所以 temp 同样接收到的是流中残存的空格字符。





Note

10.4　指针与数组

10.4.1　存储数组元素

数组，作为同名、同类型元素的有序集合，被顺序存放在一块连续的内存中，而且每个元素存储空间的大小相同。数组第一元素的存储地址就是整个数组的存储首地址，该地址放在数组名中。

对于一维数组而言，其结构是线性的，所以数组元素按下标值由小到大的顺序依次存放在一块连续的内存中。在内存中存储一维数组如图 10.14 所示。

对于二维数组而言，用矩阵方式存储元素，在内存中仍然是线性结构。

地址	元素
4001	a[0]
4005	a[1]
4009	a[2]
400C	a[3]
4011	a[4]
4015	a[5]
4019	a[6]
401C	a[7]

图 10.14　一维数组的存储

10.4.2　保存一维数组首地址

系统需要提供一定量连续的内存来存储数组中的各元素，内存都有地址，指针变量就是存放地址的变量，如果把数组的地址赋给指针变量，就可以通过指针变量来引用数组。引用数组元素有两种方法，即下标法和指针法。

通过指针引用数组，就要先声明一个数组，再声明一个指针。

```
int a[10];
int * p;
```

然后通过&运算符获取数组中元素的地址，再将地址值赋给指针变量。

```
p=&a[0];
```

把 a[0]元素的地址赋给指针变量 p，即 p 指向 a 数组的第 0 号元素，如图 10.15 所示。

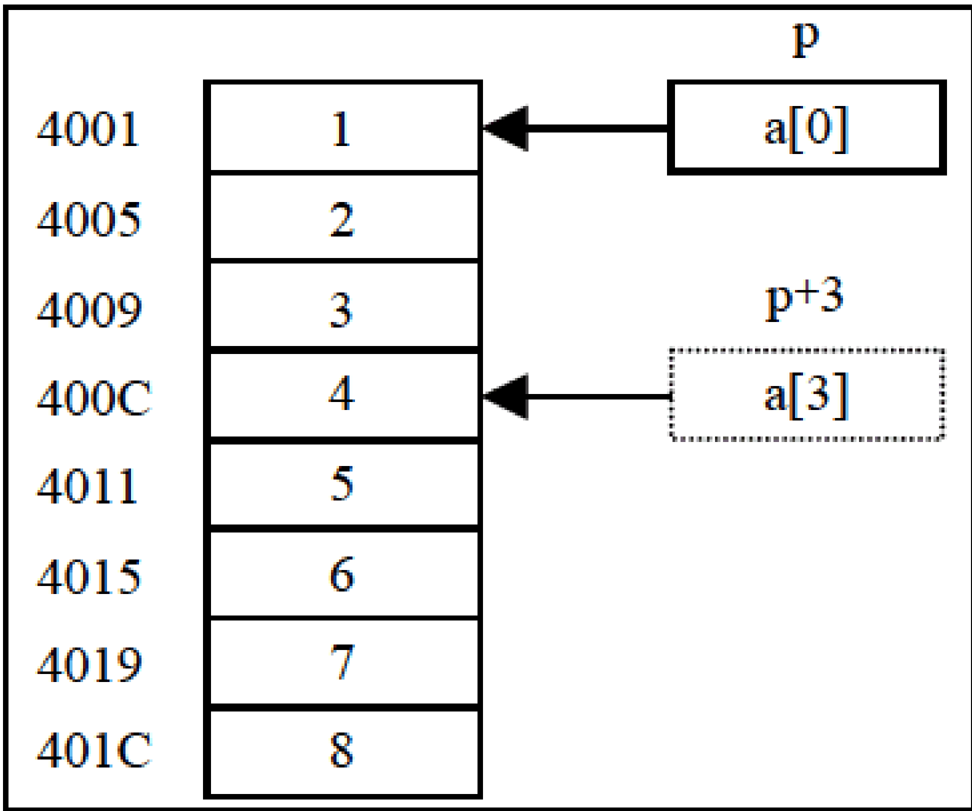
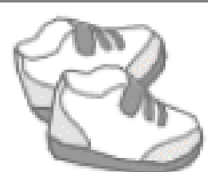


图 10.15　指针指向数组元素



下面通过实例使读者了解指针和数组间的操作，实例将实现通过指针变量获取数组中元素的功能。



Note

【例 10.9】 通过指针变量获取数组中元素。

实例位置：光盘\MR\Instance\10\10.9

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int i,a[10];
	int *p;
	for(i=0;i<10;i++)			//利用循环，分别为 10 个元素赋值
		a[i]=i;
	p=&a[0];				//让指针 p 指向数组 a 的第 0 个元素的地址
	for(i=0;i<10;i++,p++)		//将数组中的 10 个元素输出到显示设备
		cout << *p << endl;	//输出 p 指向的地址中的值
}
```

如果指针变量 p 已指向数组中的一个元素，则 p+1 指向同一数组中的下一个元素。

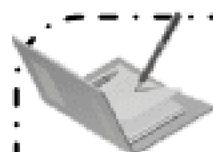
p+i 和 a+i 是 a[i]的地址。a 代表首元素的地址，a+i 也是地址，对应数组元素 a[i]。

(p+i)或*(a+i)是 p+i 或 a+i 所指向的数组元素，即 a[i]。

程序中使用指针获取数组首元素的地址，也可以将数组名赋值给指针，然后通过指针访问数组。实现代码如下：

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int i,a[10];
	int *p;
	for(i=0;i<10;i++)			//利用循环，分别为 10 个元素赋值
		a[i]=i;
	p=a;					//让 p 指向数组 a 的首地址
	for(i=0;i<10;i++,p++)		//将数组中的 10 个元素输出到显示设备
		cout << *p << endl;
}
```

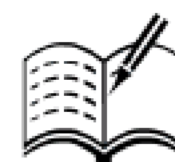
程序运行结果如图 10.16 所示。



说明：

在处理字符串函数的章节中，数组名为何能作为函数参数呢？原因如同看到的一样，它其实是一个指针常量。在数组声明之后，C++分配给了数组一个常指针，始终指向数组的第一个元素。而本章中出现的字符串处理函数中接收数组名的参数列表，也接收字符指针。关于字符串数组和指针的详细问题，在后面的章节还会讲到。





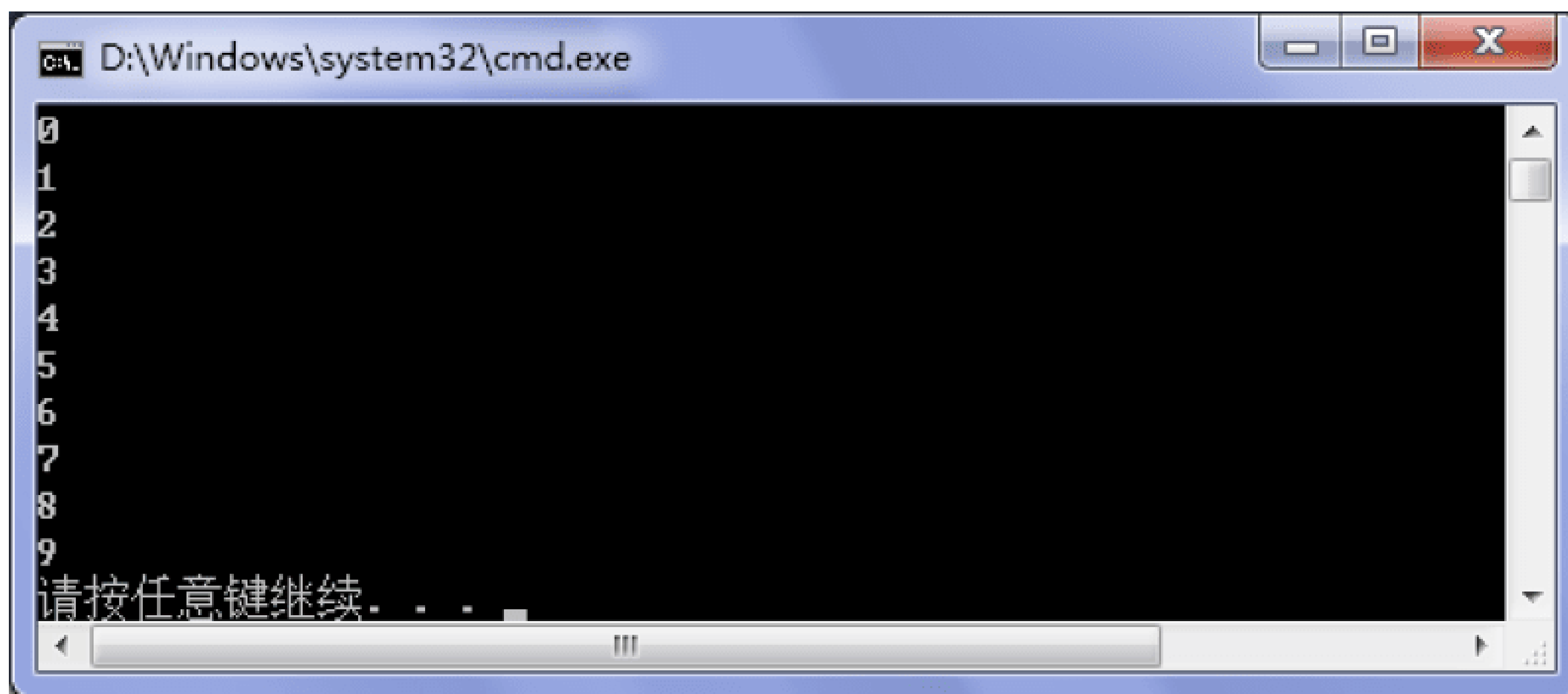


图 10.16 通过指针变量获取数组中元素

程序中使用数组地址来进行计算，a+i 表示数组 a 中的第 i 个元素，然后通过指针运算符就可以获得数组元素的值。

```
#include <iostream>
using namespace std;
void main()
{
	int i,a[10];
	int *p;
	//利用循环，分别为 10 个元素赋值
	for(i=0;i<10;i++)
		a[i]=i;
	//将数组中的 10 个元素输出到显示设备
	p=a;                              //p 指向 a 的首地址
	for(i=0;i<10;i++)
		cout << *(a+i) << endl;       //指针向后移动 i 个单位，取出其中的值并输出
}
```

指针操作数组的一些说明：

☑ *(p--)相当于 a[i--]，先对 p 进行*运算，再使 p 自减。

☑ *(++p)相当于 a[++i]，先使 p 自加，再做*运算。

☑ *(--p)相当于 a[--i]，先使 p 自减，再做*运算。

10.4.3 保存二维数组首地址

可以将一维数组的地址赋给指针变量，同样也可以将二维数组的地址赋给指针变量，因为一维数组的内存地址是连续的，二维数组的内存地址也是连续的，可以将二维数组看成是一维数组。二维数组各元素的地址如图 10.17 所示。

因为多维数组可以看成是一维数组，本例实现将多维数组转换成一维数组的功能。





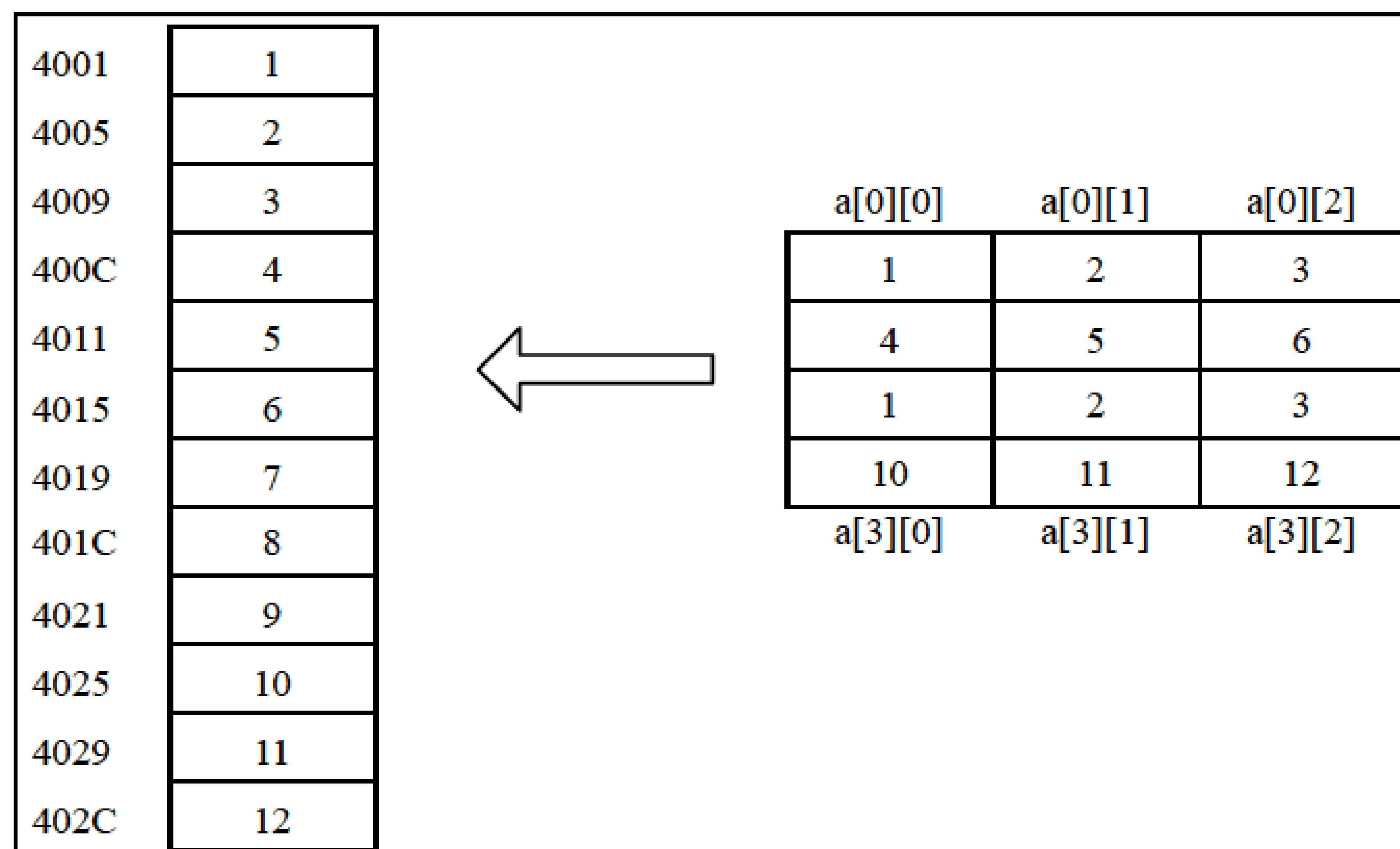


图 10.17　二维数组各元素的地址

【例 10.10】 将多维数组转换成一维数组。

实例位置：光盘\MR\Instance\10\10.10

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
	int array1[3][4]={{1,2,3,4},                    //定义 3 行 4 列整型数组并初始化
	{5,6,7,8},
	{9,10,11,12}};
	int array2[12]={0};
	int row,col,i;
	cout << "array old" <<endl;
	for(row=0;row<3;row++)                          //遍历 array1
	{
		for(col=0;col<4;col++)
		{
			cout << array1[row][col];
		}
		cout << endl;
	}
	cout << "array new" << endl;
	for(row=0;row<3;row++)                          //将 3 行合并成一行
	{
		for(col=0;col<4;col++)
		{
			i=col+row*4;
			array2[i]=array1[row][col];
		}
	}
	for(i=0;i<12;i++)                               //输出合并之后的数组
		cout << array2[i] << endl;
}
```





Note

程序运行结果如图 10.18 所示。

```
D:\Windows\system32\cmd.exe
array old
1234
5678
9101112
array new
1
2
3
4
5
6
7
8
9
10
11
12
请按任意键继续. . .
```

图 10.18　将多维数组转换成一维数组

使用指针引用二维数组和引用一维数组相同。首先声明一个二维数组和一个指针变量：

```
int a[4][3];
int * p;
```

a[0]是二维数组的第一个元素的地址，可以将该地址值直接赋给指针变量。

```
p=a[0];
```

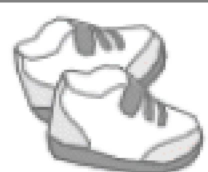
此时使用指针 p 就可以引用二维数组中的元素了。

为了更好地操作二维数组，下面通过实例来实现使用指针变量遍历二维数组的功能。

【例 10.11】 使用指针变量遍历二维数组。

实例位置：光盘\MR\Instance\10\10.11

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
using namespace std;
void main()
{
	int a[4][3]={1,2,3,4,5,6,7,8,9,10,11,12};
	int *p;
	p=a[0];
	for(int i=0;i<sizeof(a)/sizeof(int);i++)		//i<48/4，循环 12 次
	{
		cout << "address:";
		cout << a[i] ;
		cout << " is " ;
		cout << *p++ << endl;
	}
}
```





Note

程序运行结果如图 10.19 所示。

程序中通过*p 对二维数组中的所有元素都进行了引用，如果想对二维数组中某一行中的某一列元素进行引用，就需要将二维数组不同行的首元素地址赋给指针变量。如图 10.20 所示，可以将 4 个行首元素地址赋给变量 p。

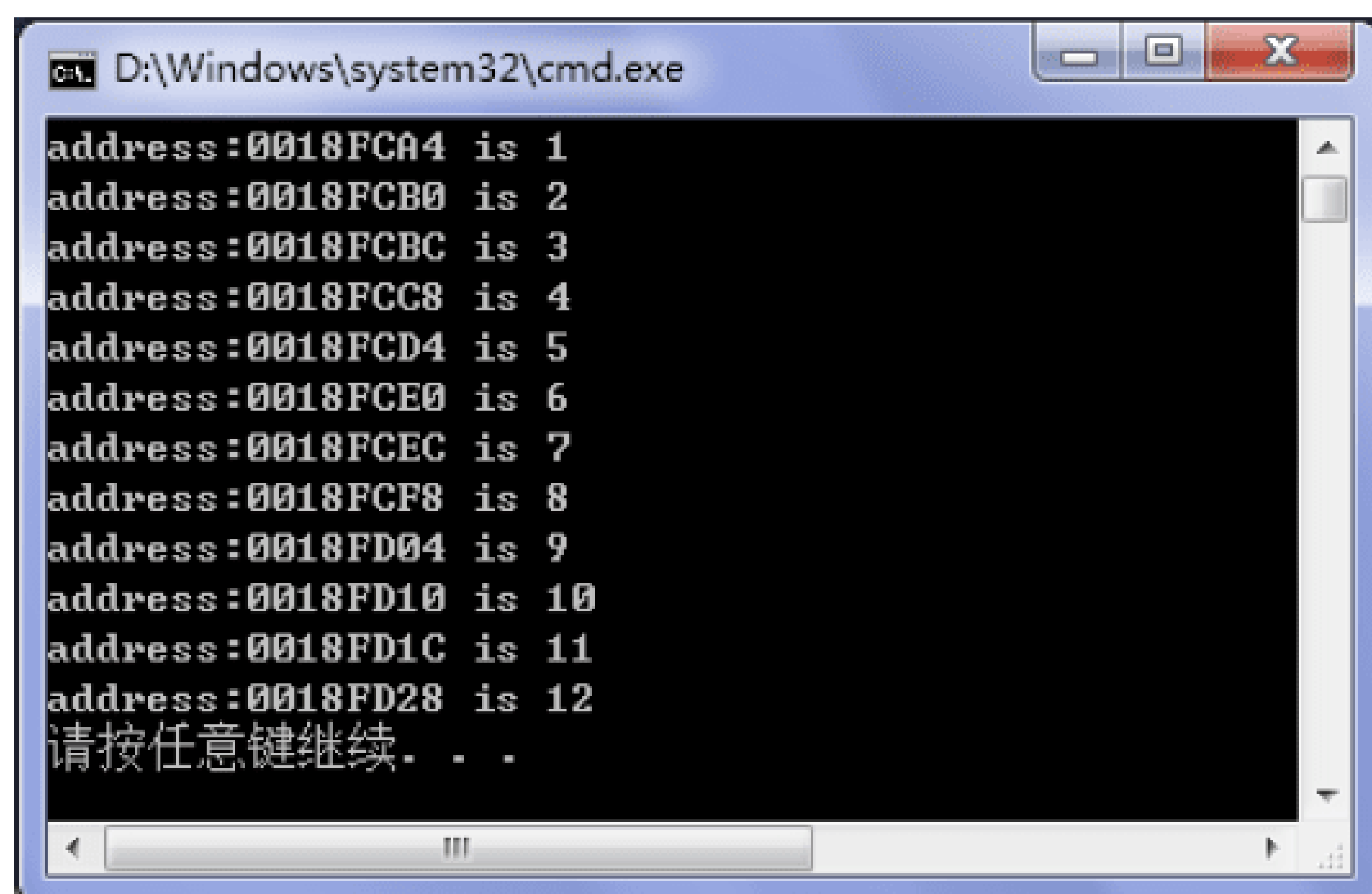


图 10.19　使用指针变量遍历二维数组

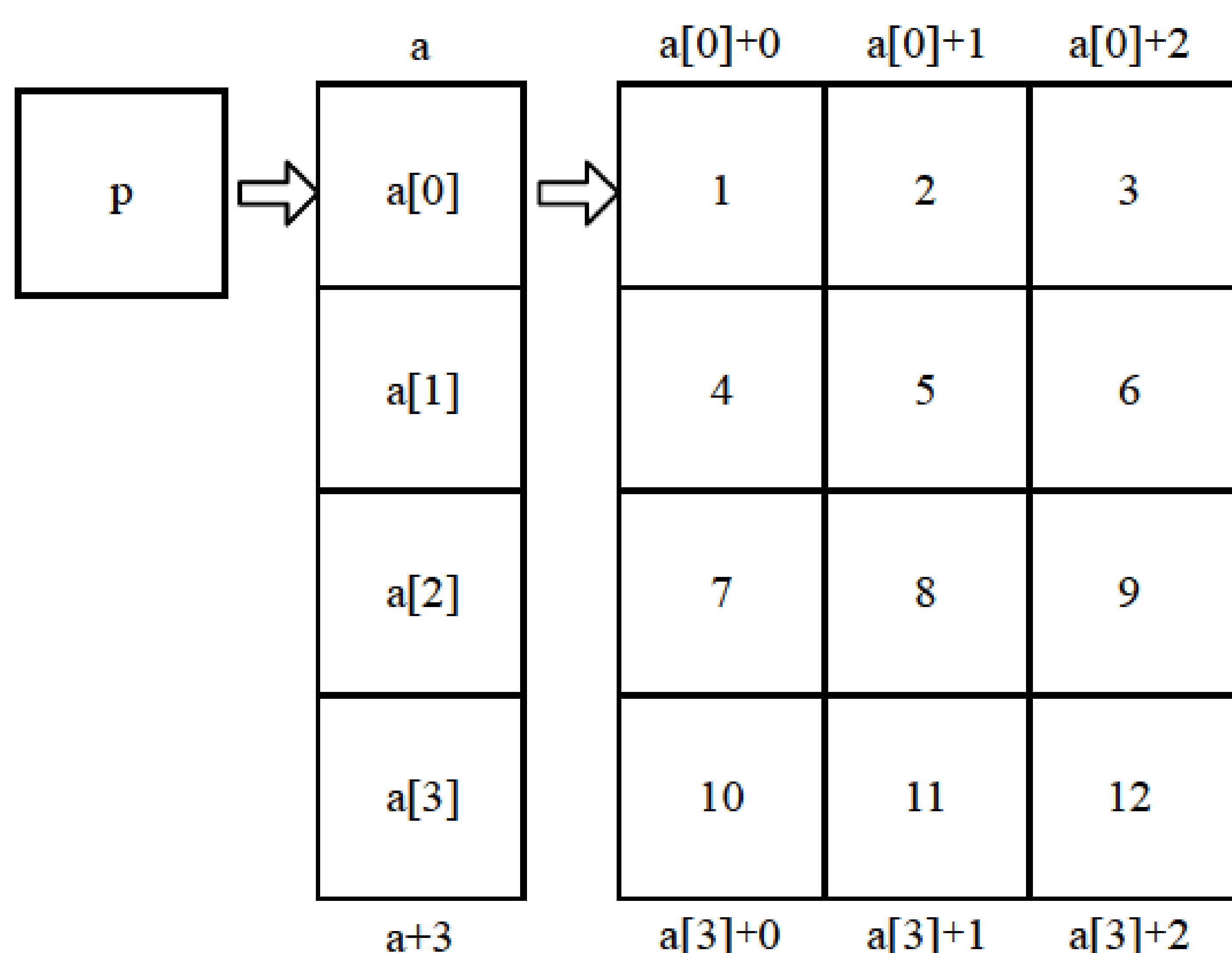


图 10.20　指针指向二维数组

a 代表二维数组的地址，通过指针运算符可以获取数组中的元素。

☑　a+n 表示第 n 行的首地址。

☑　&a[0][0]既可以看作数组 0 行 0 列的首地址，同样还可以看作二维数组的首地址。&a[m][n]就是第 m 行 n 列元素的地址。

☑　&a[0]是第 0 行的首地址，当然&a[n]就是第 n 行的首地址。

☑　a[0]+n 表示第 0 行第 n 个元素地址。

☑　*(*(a+n)+m)表示第 n 行第 m 列元素。

☑　*(a[n]+m)表示第 n 行第 m 列元素。

【例 10.12】 数组名当作指针来使用，输出二维数组的元素。

实例位置：光盘\MR\Instance\10\10.12

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
{
	int i,j;
	int a[4][3]={{1,2,3},{4,5,6},{7,8,9},{10,11,12}};
	cout << "the array is: " << endl;
	for(i=0;i<4;i++)				//4 行
	{
		for(j=0;j<3;j++)			//3 列
			cout <<*(*(a+i)+j) << endl;	//输出第 i 行的第 j 个元素
	}
}
```



程序运行结果如图 10.21 所示。

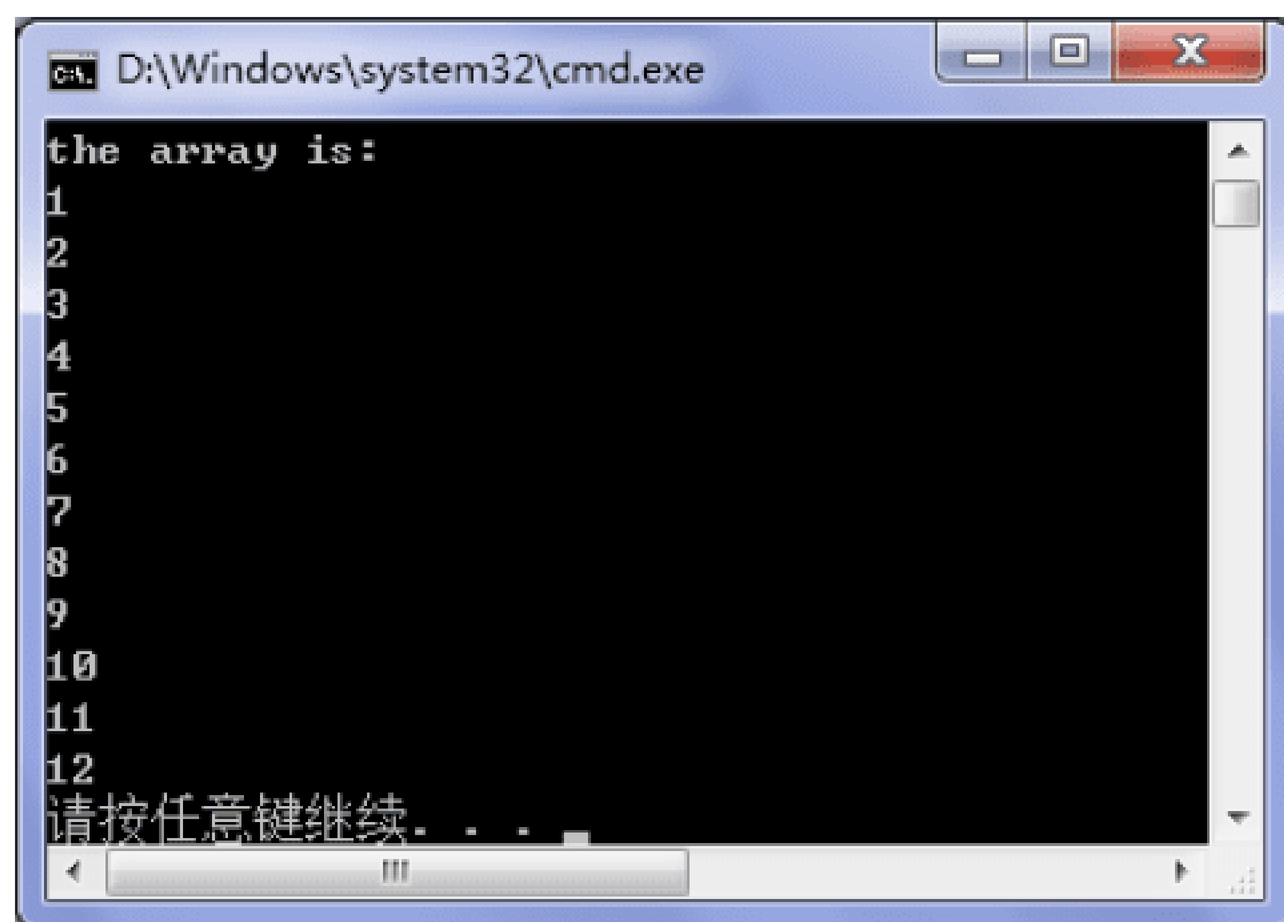
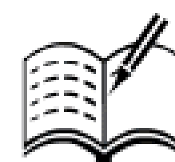


图 10.21　使用数组地址将二维数组输出



为什么指向二维数组的指针要以如此的形式表示出来？函数名 a 是一个指向数组的指针。直接依照 a 偏移 a+n 所得到的是行数组 a[n]的地址&a[n]，也就是行地址。a[n]是一个指针，它也是第 n 行的一维数组的名字。获得具体元素加上偏移求值，得到*(a[n]+m)，也就是*(*(a+n)+m)。

下面通过一个例子，使读者更好地理解二维数组的原理。

【例 10.13】 数组指针与指针数组。

实例位置：光盘\MR\Instance\10\10.13

```
#include"stdafx.h"
#include<iostream>
using namespace std;
void main()
{
	int a[3][4];
	int (*b)[4];			//定义一个数组指针，可以指向一个含有 4 个整型变量的数组
	int *c[4];			//定义一个指针数组，储存指针的数组，最多只能储存 4 个指针
	int *p;
	p = a[0];			//让 p 指向数组 a 的第 0 行的行地址
	b = a;				//让 b 指向数组 a
	cout<<"利用连续内存的特点，使用 int 指针将二维 int 数组初始化"<<endl;
	for(int i = 0;i<12;i++)		//初始化二维数组
	{
		*(p+i) = i + 1;		//给第 i 行首元素赋值
		cout<<a[i/4][i%4]<<",";
		if((i+1)%4 == 0)	//每 4 列换行
		{
			cout<<endl;
		}
	}
	cout<<"使用指向数组的指针，二维数组的值改变"<<endl;
	for(int i = 0;i<3;i++)
	{
		for(int j = 0;j<4;j++)
		{
```



```
                *(*(b+i)+j) += 10;          //通过数组指针修改二维数组内容
            }
        }
        cout<<"使用指针数组，再次输出二维数组"<<endl;
        for(int i= 0;i<3;i++)
        {
            for(int j = 0;j<4;j++)
            {
                c[j] = &a[i][j];            //用指针数组里的指针指向 a[i][j]
                cout<<*(c[j])<<",";
                if((j+1)%4 == 0)            //每 4 列换行
                {
                    cout<<endl;
                }
            }
        }
}
```



Note

程序运行结果如图 10.22 所示。

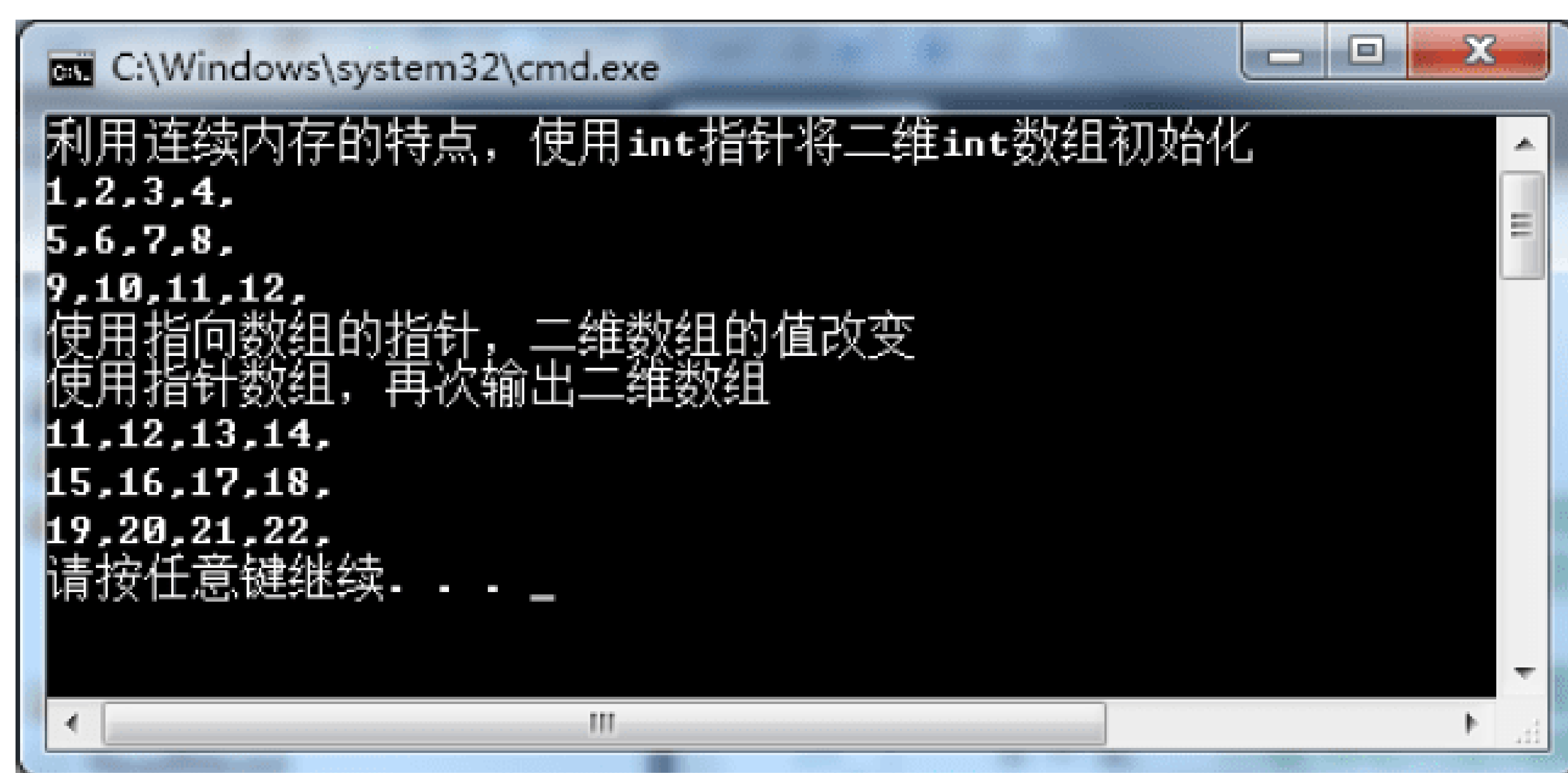


图 10.22　执行结果

10.4.4　指针与字符数组

字符数组是一个一维数组，使用指针同样也可以引用字符数组。引用字符数组的指针为字符指针，字符指针就是指向字符型内存空间的指针变量，其一般的定义语句如下：

```
char *p;
char *string="www.mingri.book";
```

【例 10.14】 通过指针偏移连接两个字符串。

实例位置：光盘\MR\Instance\10\10.14

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main()
```





```
{
    char str1[50],str2[30],*p1,*p2;
    p1=str1;                    //让两个指针分别指向两个数组
    p2=str2;
    cout << "please input string1:"<< endl;
    gets(str1);                 //给 str1 赋值
    cout << "please input string2:"<< endl;
    gets(str2);                 //给 str2 赋值
    while(*p1!='\0')
    p1++;                       //把 p1 移动到 str1 的末尾
    while(*p2!='\0')
    *p1++=*p2++;                //取 p2 指向的值赋到 p1 指向的地址（str1 的末尾），即连接 str1 和 str2
    *p1='\0';
    cout << "the new string is:"<< endl;
    puts(str1);                 //输出新的 str1
}
```

程序运行结果如图 10.23 所示。

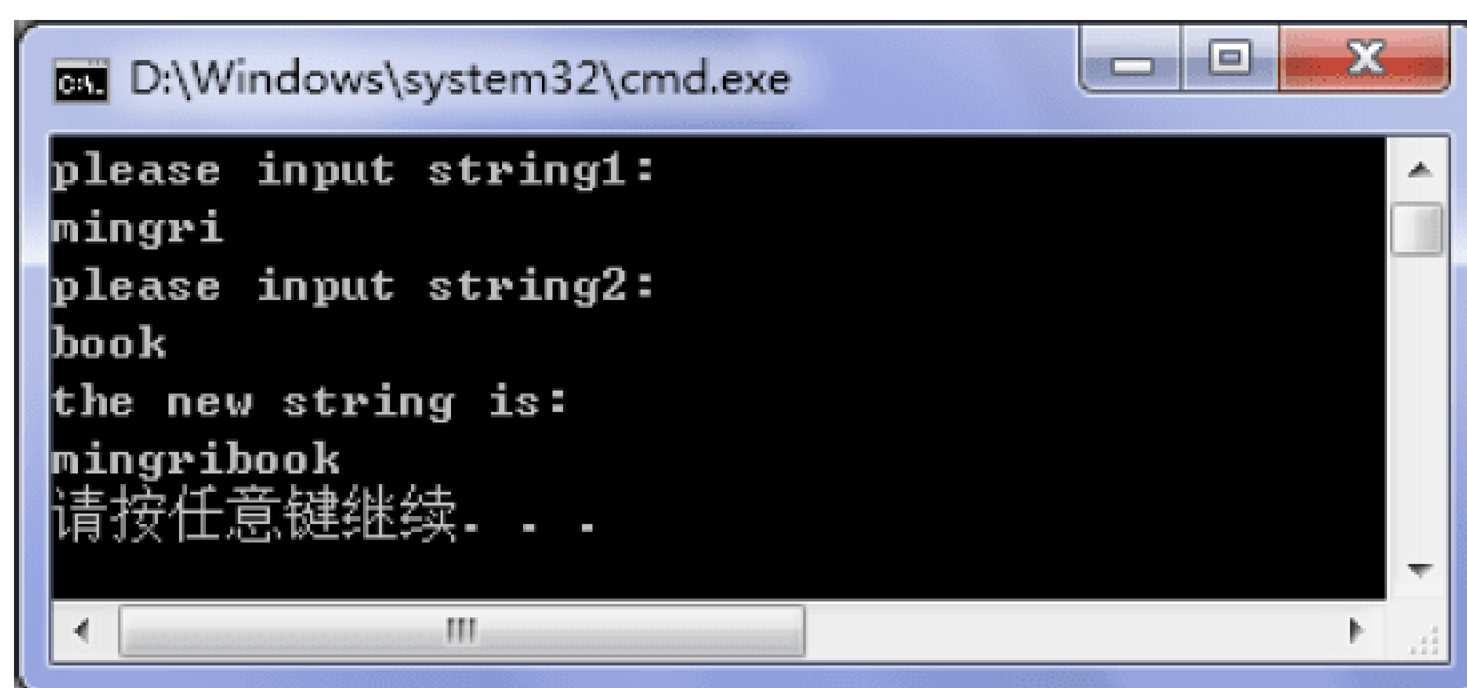


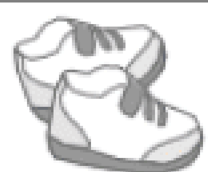
图 10.23　连接两个字符数组

同样还可以使用处理字符串函数 strcat 来实现上面的例子。

【例 10.15】 通过字符串函数 strcat 连接两个字符串。

实例位置：光盘\MR\Instance\10\10.15

```
#include "stdafx.h"
#include <iostream>
using namespace std;
void main()
{
    char str1[50],str2[30],*p1,*p2;
    p1=str1;
    p2=str2;
    cout << "please input string1:"<< endl;
    gets(str1);
    cout << "please input string2:"<< endl;
    gets(str2);
    strcat(str1,str2);          //连接 str1 和 str2
    cout << "the new string is:"<< endl;
    puts(str1);
}
```





Note

程序运行结果如图 10.24 所示。



图 10.24　使用 strcat 连接两个字符串

10.4.5　数组作函数参数

在函数调用过程中，有时需要传递多个参数，如果传递的参数都是同一类型的则可以通过数组的方式来传递参数，作为参数的数组可以是一维数组，也可以是多维数组。使用数组作函数参数最典型的就是 main 函数。带参数的 main 函数形式如下：

```
main(int argc,char *argv[])
```

main 函数中的参数可以获取程序运行的命令参数，命令参数就是执行应用程序时后面带的参数。例如，在 CMD 控制台执行 dir 命令，可以带上/w 参数，dir/w 命令是以多列的形式显示出文件夹内的文件名。main 函数中参数 argc 是获取命令参数的个数，argv 是字符指针数组，可以获取具体的命令参数。

【例 10.16】 获取命令参数。

实例位置：光盘\MR\Instance\10\10.16

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void main(int argc,char *argv[])
{
	cout << "the list of parameter:" << endl;
	while(argc>1)                    //获取的参数个数比 1 大时，输出参数的内容
	{
		++argv;
		cou << *argv << endl;
		--argc;
	}
}
```

上面代码会在项目文件夹中生成 exe 文件。通过本书附带光盘的路径可以找到该程序的文件夹。打开其中的 DEBUG 文件夹就可以找到。将它复制到本地的文件夹中，使用控制台来运行它，如图 10.25 所示。





Note

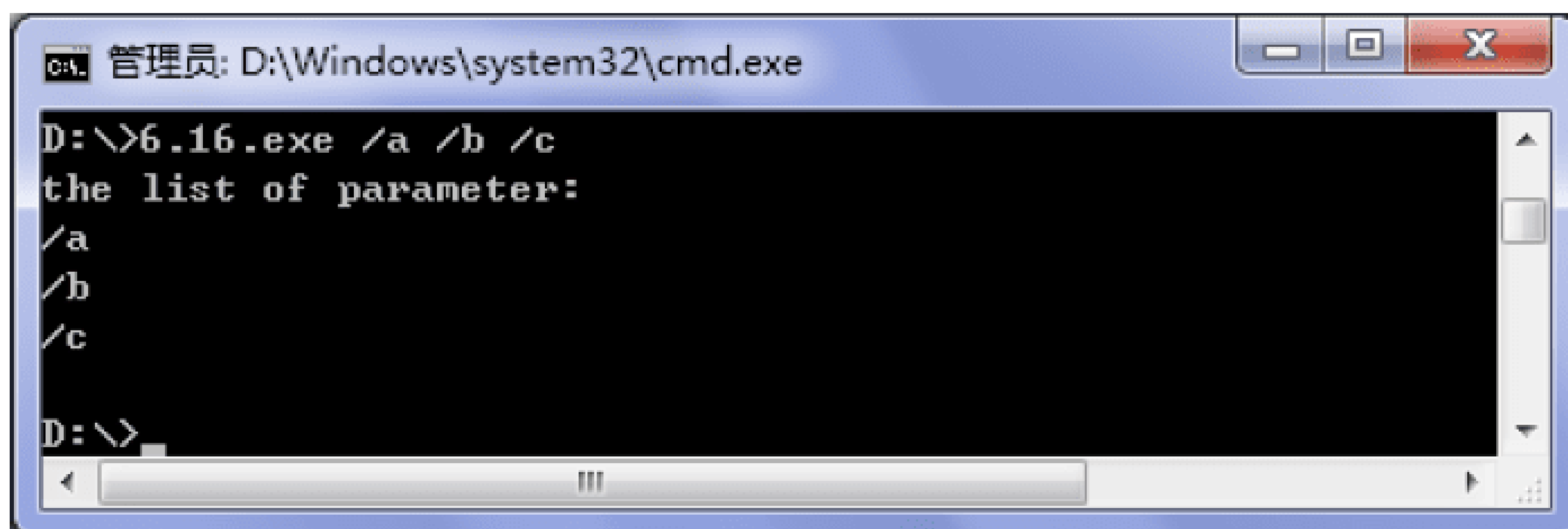


图 10.25　获取命令参数

程序执行时输入命令参数“/a /b /c”，程序运行以后将 3 个命令参数输出，每个参数都是以空格隔开，应用程序后有 3 个空格，代表程序有 3 个命令参数，argc 的值就为 3。

二维数组在作为函数参数时，可以将二维数组转换成一个一维的指针数组。main 函数中的 argv 数组就可以是一个二维的字符数组。

【例 10.17】 输出每行数组中的最小值。

实例位置：光盘\MR\Instance\10\10.17

```
#include "stdafx.h"
#include<iostream>
using namespace std;
void mix(int (*a)[4],int m)                        //进行比较和交换的函数
{
    int value,i,j;
    for(i=0;i<m;i++)
    {
        value=*(*(a+i));                           //第 i 行第 0 个元素
        for(j=0;j<4;j++)
            if(*(*(a+i)+j)<value)
                value=*(*(a+i)+j);                 //第 i 行第 j 个元素
        cout <<"第" << i+1<<"行" ;
        cout <<":最小值" << value << endl;         //输出最小值
    }
}
void main()
{
    int a[3][4],i,j;
    int (*p)[4];                                   //定义数组指针
    p=&a[0];
    for(i=0;i<3;i++)                               //给数组赋值
    {
        cout << "请输入第:"<<i+1<<"行"<< endl;
        for(j=0;j<4;j++)
        {
            cin >> a[i][j];
        }
    }
    mix(p,3);                                      //调 mix 函数
}
```





Note

程序运行结果如图 10.26 所示。

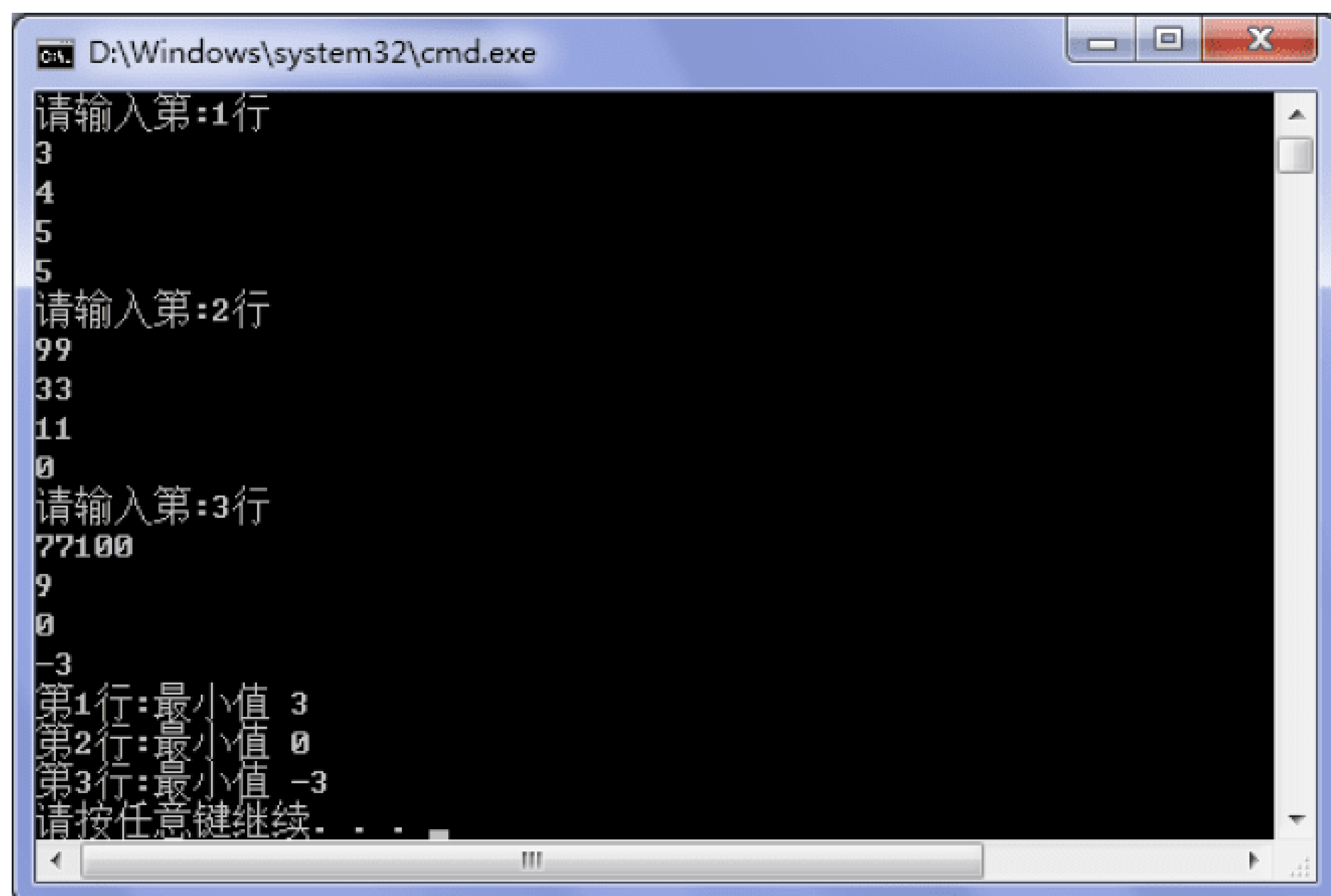


图 10.26　输出每行数组中的最小值

程序需要用户输入 12 个数值来作为一个 3 行 4 列数组的元素，然后按行进行比较，输出每行最小元素。*(a+i)代表数组每行第一个元素，*(*(a+i)+j)代表数组指定行中的某个列元素。函数 mix 对数组每行元素逐一进行比较，将最小值赋给变量 value，然后输出变量 value 的值。

10.4.6　动态分配数组

有时在获得一定的信息之前，我们并不确定数组的大小。例如，记录本日学生考试成绩，又如获得某旅社的当日旅客名单。动态分配数组则可以使用变量作为数组大小，使数组的大小符合要求。

【例 10.18】 动态获得斐波纳契数列。

实例位置：光盘\MR\Instance\10\10.18

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int main()
{
	int k=1;
	cout<<"请输入斐波纳契数列最大阶数"<<endl;
	while(cin>>k,!(2<k))
	{
		cout<<"请输入大于 2 的数字"<<endl;
	}
	//int a[k];                              //不能使用变量申请栈内存，注释掉
	int *pArray = new int[k];                //动态数组创建，堆内存已经分配完毕
	*pArray = 1;                             //斐波纳契数列的第一项与第二项为 1
	*(pArray+1) =1;                          //以上两个语句为指针形式的数组表示方法
	for(int i =2;i<k;i++)
```





```
    {
        pArray[i]=pArray[i-2]+pArray[i-1];            //数组正常的表示方法
    }
    cout<<"请输入您想要获得的斐波纳契数列项的阶数"<<endl;
    int i = 0;                                        //循环体外的 i
    while(cin>>i,!(0<i&i<k+1))
    {
        cout<<"请输入介于 1 到"<<k<<"之间的数字"<<endl;
    }
    cout<<"斐波纳契第"<<i<<"项为:"<<pArray[i-1]<<endl;
    delete []pArray;
    return 0;
}
```

程序运行结果如图 10.27 所示。

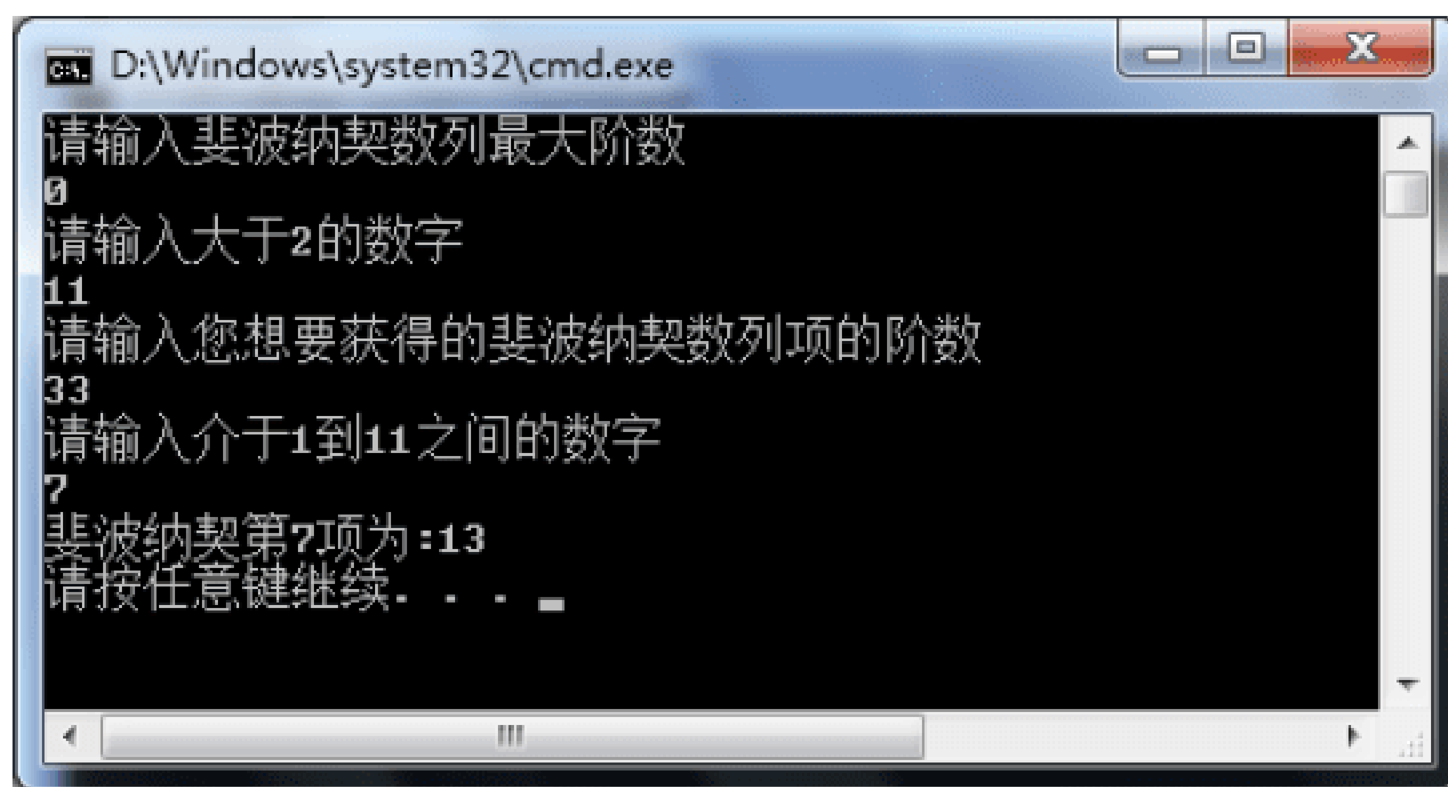
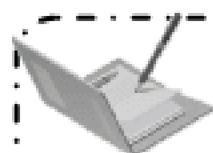


图 10.27　动态获得斐波纳契数列

根据输入数字动态创建了一个数组，使它包含了斐波纳契数列的前 k 项。



说明：

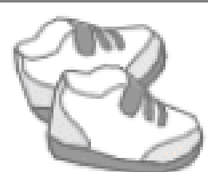
斐波纳契数列是遵循这样规律的一组数。第一项与第二项为 1，以后各个项都等于前面两项的和。

10.5　字符串类型

标准库模板库 STL 提供给我们一种自定义类型数据 string，使我们能更好地操作字符串。

10.5.1　使用本地字符串类型 string

与标准输入/输出流一样，引入 string 类型数据需要添加相应的头文件和使用相应的名字空间标识。string 类型所在的头文件是 string.h，可以通过执行#include<string>命令让 Visual Studio 2010 编译器链接它。





声明一个 string 变量，形式如下：

```
std::string s;
```

初始化 string 类型的变量有多种形式：

```
std::string s1("字符串");
std::string s2 = "字符串";
std::string s3 = (3,'A');    //s3 的内容为 AAA
```

通过“[ ]”号可以对 string 字符串相应位置的字符进行访问和修改。

【例 10.19】 修改 string 字符串的单个字符。

实例位置：光盘\MR\Instance\10\10.19

```
#include "stdafx.h"
#include <string>
#include <iostream>
using namespace std;
int main()
{
    string s = "Good Morning!";
    cout<<s<<endl;
    cout<<"访问并修改第 5 个字符"<<endl;
    s[4] = '!';
    cout<<s<<endl;
    return 0;
}
```

程序运行结果如图 10.28 所示。



图 10.28　修改单个字符

10.5.2　连接 string 字符串

使用+号可以将两个 string 字符串连接起来。同时，string 还支持标准输入/输出函数。

【例 10.20】 连接两个 string 字符串。

实例位置：光盘\MR\Instance\10\10.20

```
#include "stdafx.h"
#include <iostream>
```





```
#include <string>
using namespace std;
int main()
{
	string str1 = "您好,";
	string str2;
	cout<<"请输入您的姓名:"<<endl;
	cin>>str2;
	str1 = str1+str2;			//连接两个字符串
	string str3 = "!明日科技为您服务。";
	str1 += str3;				//str1 = str1 + str3
	cout<<str1<<endl;
	return 0;
}
```

程序运行结果如图 10.29 所示。



图 10.29　string 字符串的连接

10.5.3　比较 string 字符串

使用>、!=、>=等比较运算符可以比较两个字符串的内容。比较的方法是将两个 string 字符串从头开始比较每一个字符，直到出现两者不一致。比较这两个不相同的字符的字面值，得出相应的结果。

【例 10.21】　比较两个 string 字符串的大小。

实例位置：光盘\MR\Instance\10\10.21

```
int main(int argc, _TCHAR* argv[])
{
	string s1;
	string s2;
	cout<<"请输入两个字符串"<<endl;
	cin>>s1;
	cin>>s2;
	if(s1 == s2)
	{
		cout<<"两个字符串相等"<<endl;
	}
	else if(s1>s2)
	{
```



```
		cout<<"第一个字符大于第二个字符串"<<endl;
	}
	else
	{
		cout<<"第二个字符大于第二个字符串"<<endl;
	}
	return 0;
}
```



Note

程序运行结果如图 10.30 所示。

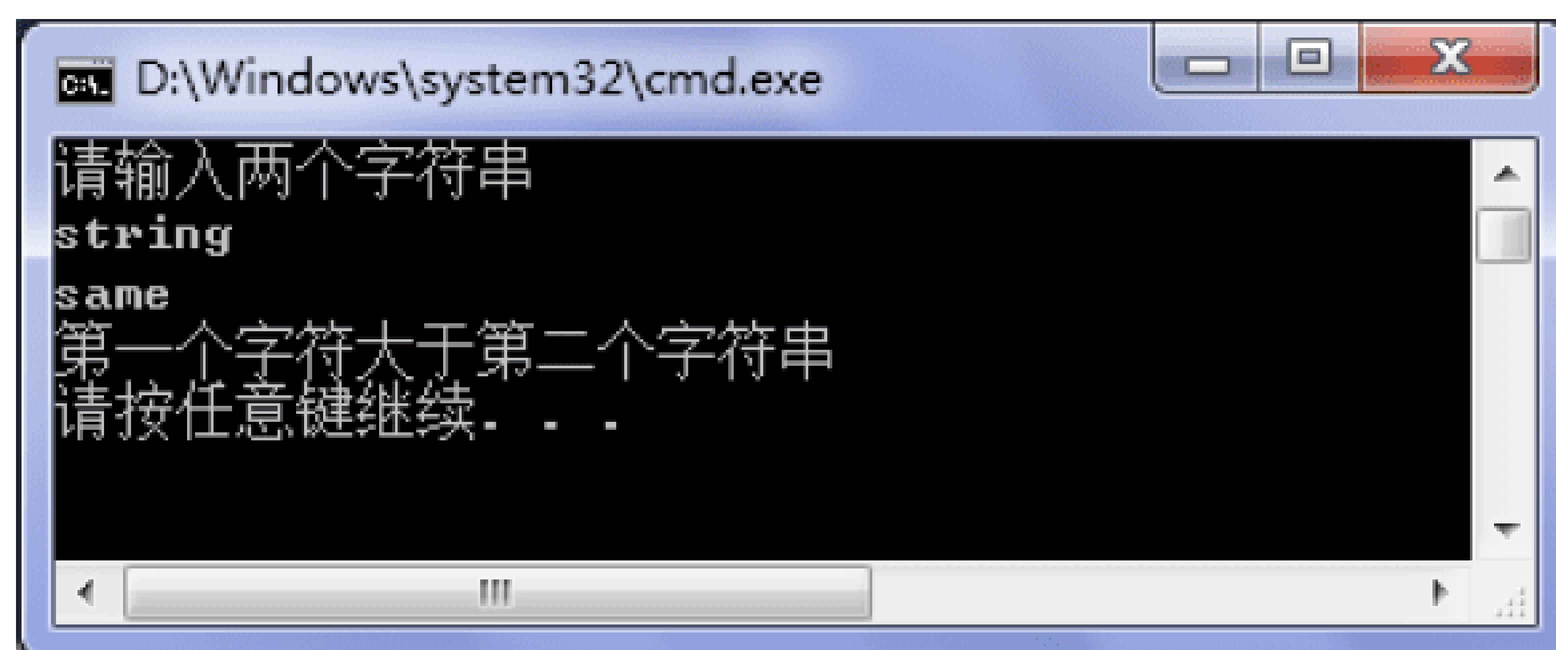


图 10.30　比较两个字符串

输入两个字符串 string 和 same，它们从第二个开始出现不一致，“t”的 ASCII 码要大于“a”，所以“string”大于“same”。

10.5.4　定义 string 类型数组

数组中存储的数据也可以是 string 类型的，例如：

【例 10.22】 string 类型的数组。

实例位置：光盘\MR\Instance\10\10.22

```
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;
int main(int argc, _TCHAR* argv[])
{
	string sArrary[5] = {"明日","科技","为","您","服务！！ "};
	string s="";	//空的 string
	for(int i = 0;i<5;i++)
	{
		s+=sArrary[i];
	}
	cout<<s<<endl;
	return 0;
}
```

数组中储存了 5 个 string 对象，将它们拼接起来，如图 10.31 所示。



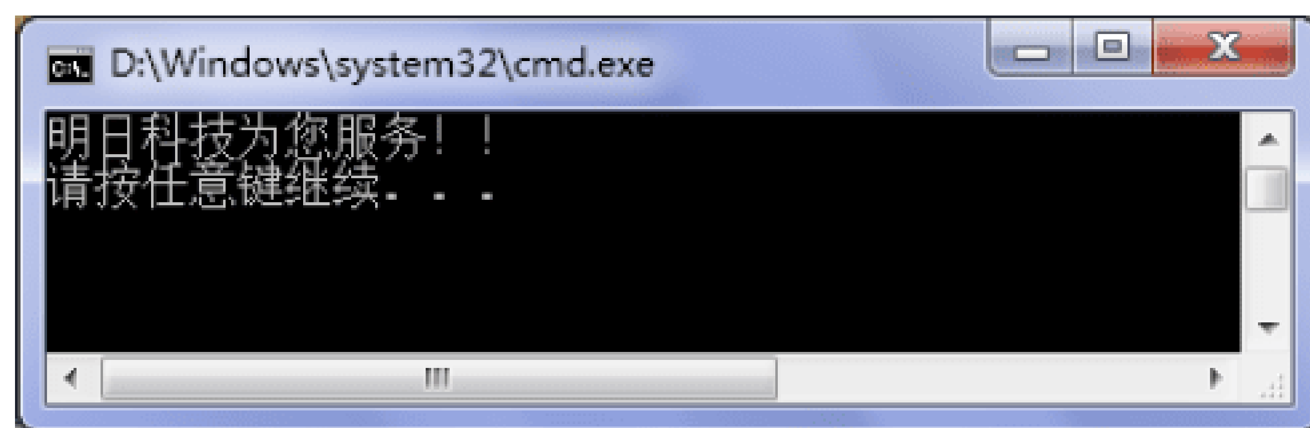


图 10.31　string 类型数组的拼接



string 与字符串数组都可以表示一段字符串，但它们有着很大的区别：

☑　类别不同。

☑　字符串数组需要防范越界、结束符等问题，而 string 不需要。

☑　字符串可以通过地址的形式通过=赋值给 string，但 string 不能直接赋值给字符串数组。

【例 10.23】 用字符串数组给 string 类型数组赋值。

实例位置：光盘\MR\Instance\10\10.23

```
#include "stdafx.h"
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;
int main(int argc, _TCHAR* argv[])
{
	char aArray[8] ="Welcome";
	string s = aArray;
	cout<<s<<endl;
	s = &aArray[2];
	cout<<s<<endl;
	return 0;
}
```

string 对象 s 通过 aArray 的数组名初始化为 welcome，之后将 aArray 第三个元素的地址通过赋值符号=传递给 s，输出结果如图 10.32 所示。



图 10.32　字符串数组地址赋值于 string

10.6　综 合 应 用

10.6.1　名字排序

【例 10.24】 本实例设计一个程序，将一组英文名字按照字母顺序依次储存到一个数组中，



然后输出它们。该程序具有比较字符串相应位置字母、存放字符串和输出的功能。先比较字符串的首字母，若相同，则继续比较下一位字母，若不同，交换字符串，按 ASCII 码从小到大排序。代码如下：



Note

实例位置：光盘\MR\Instance\10\10.24

```
#include "stdafx.h"
#include <string>
#include <iostream>
using namespace std;
int main(int argc, _TCHAR* argv[])
{
	string sArray[5] ={"Mike","Andy","Tom","Jack","Mary"};
	string temp;
	for(int i = 0;i<4;i++)
	{
		for(int j = i+1;j<5;j++)
		{
			if(sArray[i]>sArray[j])
			{
				temp = sArray[i];
				sArray[i] =sArray[j];
				sArray[j] = temp;
			}
		}
	}
	for(int i = 0;i<5;i++)
	{
		cout<<sArray[i]<<endl;
	}
	return 0;
}
```

程序运行结果如图 10.33 所示。

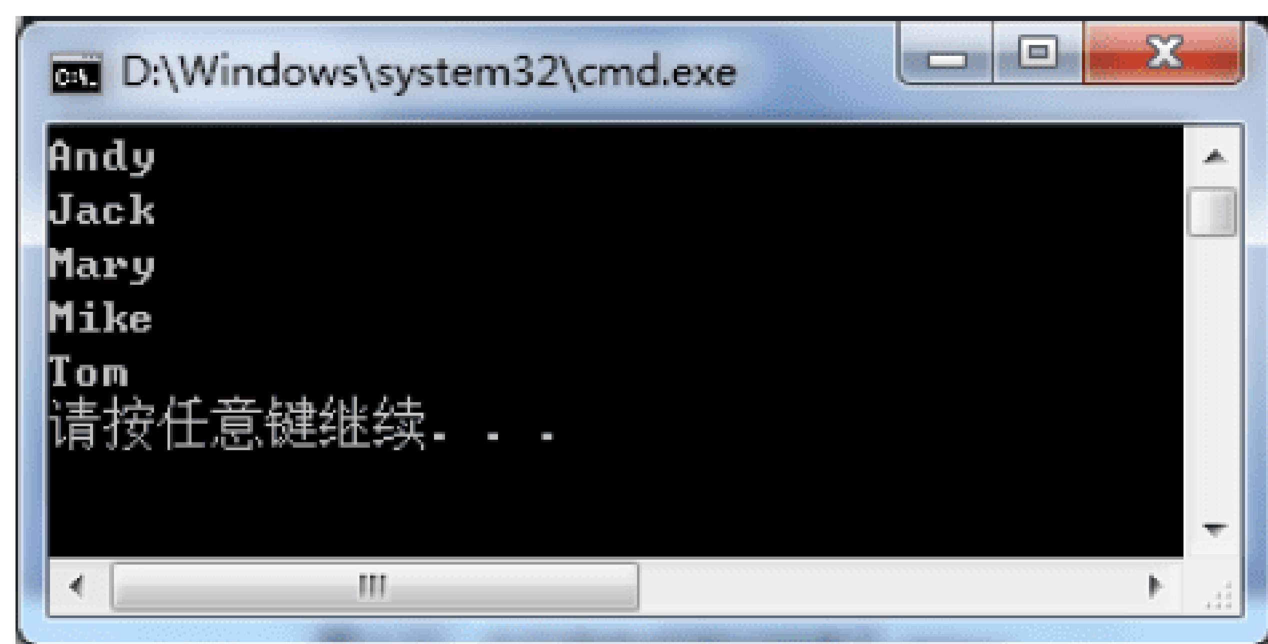


图 10.33　名字排序

10.6.2　查找数字

【例 10.25】 本实例设计程序，实现随机输入 20 个正整数，记录这些数当中能被 7 整除的数字并输出它们的值和输入顺序。

实现过程：定义一个数组用来存储输入的数字，判断输入的数字，如果满足条件，则存入数组中，否则给数组中相应位置赋值为 0，最后判断数组中不为 0 的即是要求的数字。代码如下：

实例位置：光盘\MR\Instance\10\10.25

```
#include "stdafx.h"
#include <iostream>
```






```
using namespace std;
int main(int argc, _TCHAR* argv[])
{
	cout<<"请输入 20 个正整数:"<<endl;
	int dArray[20];                          //定义一个整型数组
	int temp = 0;                            //定义一个临时变量
	for(int i = 0;i<20;i++)                  //循环输入 20 次
	{
		cin>>temp;
		if(temp%7 ==0)                       //判断输入的数字是否能被 7 整除
		{
			dArray[i] = temp;                //若整除，将此数字存入相应位置
		}
		else
		{
			dArray[i] = 0;                   //否则赋 0 值
		}
	}
	for(int i = 0;i<20;i++)                  //循环遍历数组
	{
		if(dArray[i]!=0)                     //找到上面记录的不为 0 的数字，即能被 7 整除
		{
			cout<<"第"<<i+1<<"个数是 7 的整数倍，它的值为:"<<dArray[i]<<endl;
		}
	}
	return 0;
}
```

程序运行结果如图 10.34 所示。

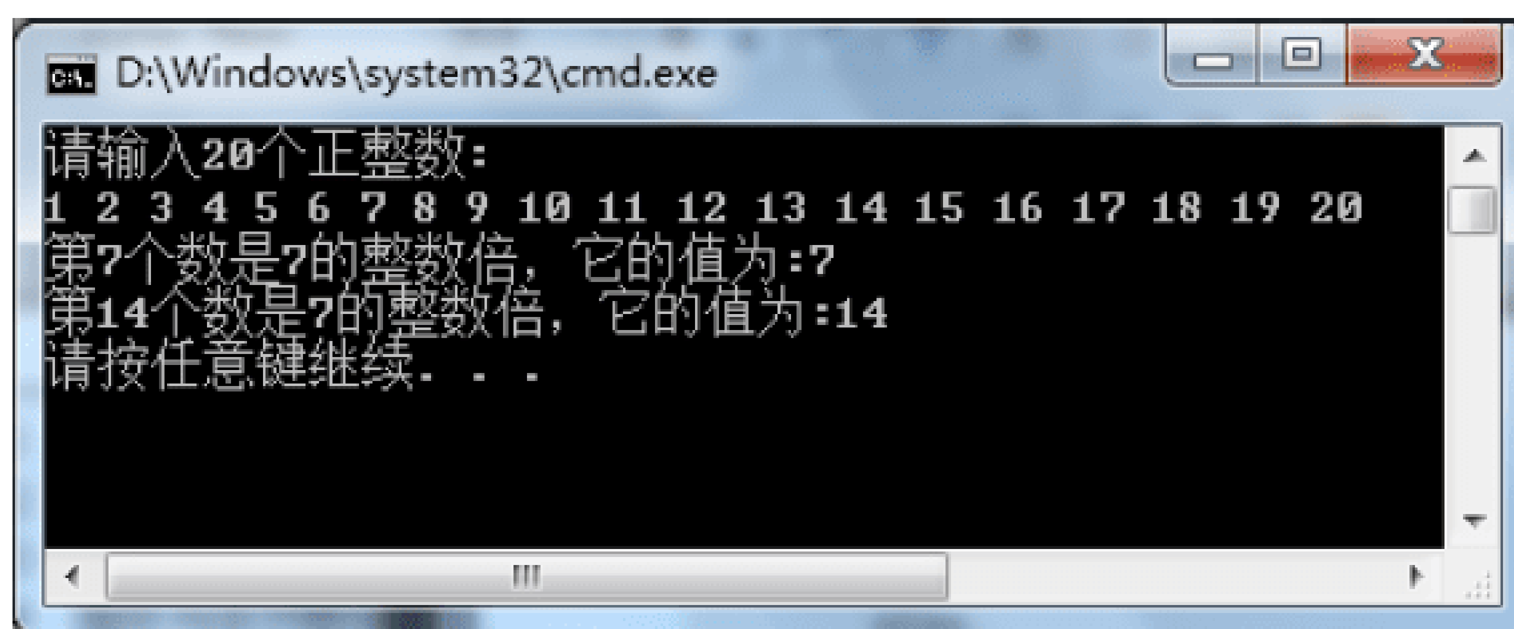


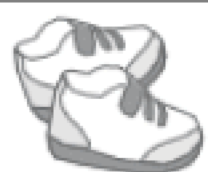
图 10.34　查找数字

10.6.3　求平均身高

【例 10.26】 编写一个函数，实现求平均身高的功能，输入学生人数和身高，计算平均身高并输出。代码如下：

实例位置：光盘\MR\Instance\10\10.26

```
#include "stdafx.h"
#include <iostream>
```





```
using std::cin;
using std::cout;
using std::endl;
void average(float &aver, float *p_height, int num)
{
    float sum = 0;
    for(int i = 0; i < num; i++)
        sum += *p_height++;                    //累加每个人的身高，求和
    aver = sum / num;                          //求平均身高
}
int main(int argc, _TCHAR* argv[])
{
    float height[100],aver;
    int num;
    cout<<"please input the number of students:\n";
    cin>>num;
    printf("please input student`s height:\n");
    for(int i = 0; i < num; i++)
        cin>>height[i];
    average(aver, height, num);                //调用 average 函数
    cout<<"average height is "<<aver<<endl;
    return 0;
}
```

程序运行结果如图 10.35 所示。

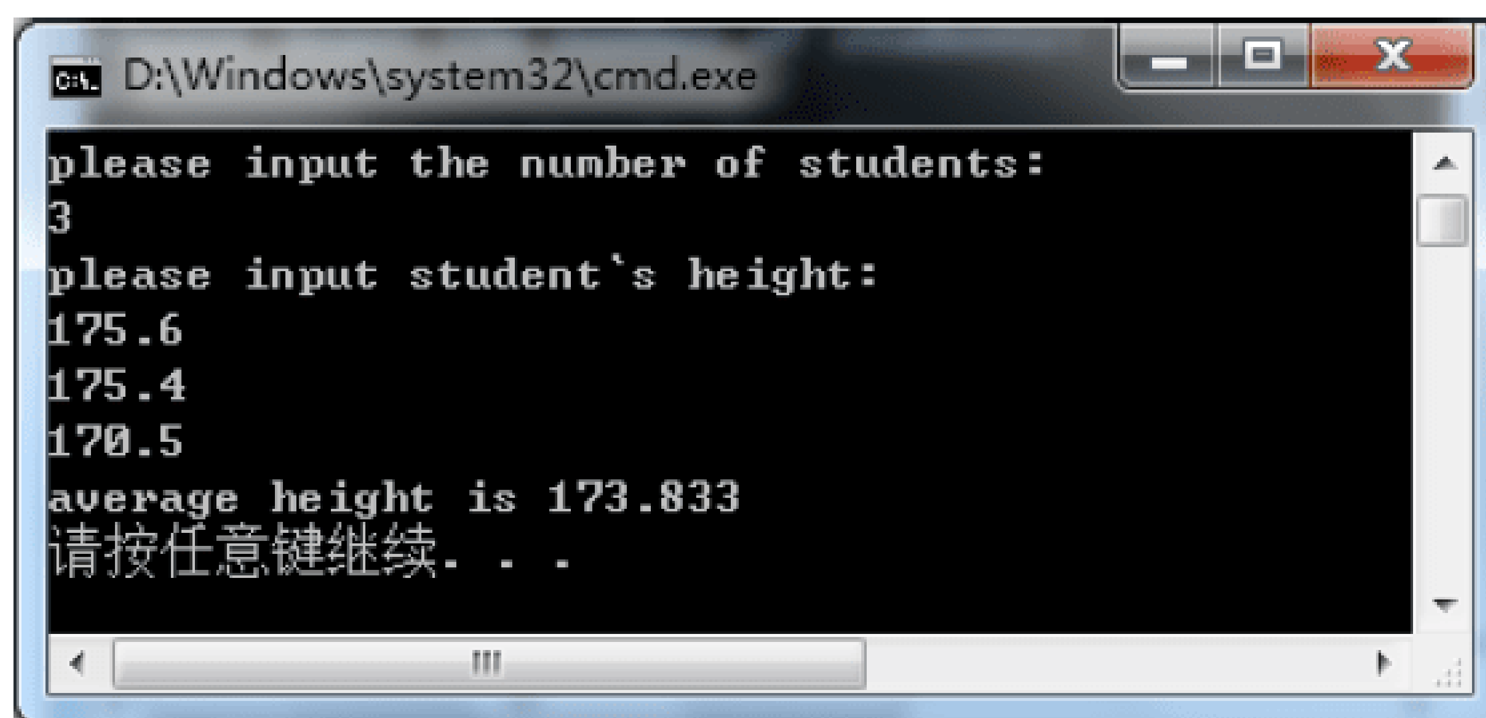


图 10.35　求平均身高

10.7　本章常见错误

10.7.1　不能对数组名直接赋值

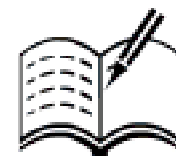
数组名是常量，不能给常量赋值，所以对数组名赋值是错误的，只能用 strcpy 函数。例如：

```
char str[4];
str = "abc";              //错误
strcpy(str,"abc");        //正确
```



数组名也不能做自加运算，变量才可以自加。例如：

```
str++;                    //错误
```



但是数组名可以做加法操作，如加上一个整数。此时相当于把数组名当作指针来用，指向 str[0]，加 1 相当于指针向后偏移一个字节，指向 str[1]。例如：

```
cout<<*str;               //输出 a
cout<<*(str+1)            //输出 b
```

10.7.2　sizeof(a)和 sizeof(a+1)

有如下代码，分别求 sizeof(a)和 sizeof(a+1)的计算结果：

```
int a[10]="123";
```

在表达式 sizeof(a)中，数组名 a 代表数组本身，所以此时 sizeof(a)测出的是整个数组的大小，是 10 个 int 型变量所占空间，4*10 = 40。不要误以为它的大小是 10。同理，如果是 double 类型，就是 80。

sizeof(a+1)测出的是指针类型的大小。此时的 a+1 意味着 a 被当作指针处理。在 32 位系统中指针的大小是 4，故 sizeof(a+1)的测量结果是 4。

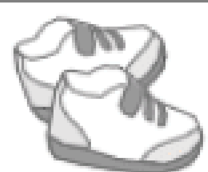
10.7.3　注意区分数组指针和指针数组

数组指针是一个指针，指向一个数组的整体地址，该指针的类型决定它所能指向的数组的类型。如“int (*p)[10];”是一个数组指针，它可以指向一个含有 10 个整型元素的数组。

指针数组是一个数组，它里面包含若干个类型相同的指针变量。如“int *p[10];”是一个指针数组，它里面有 10 个能够指向 int 类型数据的指针变量。

10.8　本 章 小 结

本章详细讲解了数组、指针的概念和使用方法。数组是以连续的方式储存，能与指针很好地配合使用。二维数组可以看作数据类型是数组的数组，它的储存方式也是连续的，善用这个特性可以很好地解决需要遍历数组的问题。字符串是人机交互，数据传递中必不可少的数据类型。它的实质是字符型一维数组。最后介绍的 string 自定义类型数据，相比字符数组使用会方便一些，但占用空间要相对较大，也被称为 string 类。



10.9 跟 我 上 机



参考答案：光盘\MR\跟我上机

设计一个程序，实现字符串复制功能（不直接调用库函数 strcpy），将 p 指向的字符串复制到数组 str 中，并输出显示。实现如下：

```
#include <iostream>
#define STRING "abcdefg"                    //定义字符串宏
#define LENGTH 10                           //定义数组大小宏
using namespace std;
void my_Copy(char *p,char *str)             //自定义复制函数
{
    int i = 0;
    while('\0' != (str[i++] = *p++));       //给 str 元素逐个赋值
}
int main()
{
    char *p = STRING;
    char str[LENGTH];
    cout<<"p 指向的字符串：\n"<<p<<endl;
    my_Copy(p,str);                         //调用自定义复制函数
    cout<<"复制到数组 str 中的字符串：\n"<<str<<endl;
    return 0;
}
```

第 2 篇



提高篇

- 第 11 章　面向对象编程
- 第 12 章　从基类到派生类
- 第 13 章　C++模板的使用
- 第 14 章　代码整理
- 第 15 章　掌握 C++标准模板库
- 第 16 章　利用文件处理数据

第11章

面向对象编程

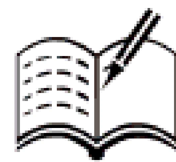
（视频讲解：1小时20分钟）

面向对象的设计思想在当前已经被广泛承认和应用。面向对象编程可以有效解决代码复用问题，它不同于以往的面向过程编程，面向过程编程需要将功能细分，而面向对象需要将不同功能抽象到一起。类是对象的实现，面向对象中的类是抽象概念，而类是程序开始过程中定义一个对象，用类定义对象可以是现实生活中的真实对象，也可以是从现实生活中抽象的对象。

本章能够完成的主要范例（已掌握的在方框中打勾）

- [ ] 通过实例化的对象访问类成员
- [ ] 利用构造函数对成员变量赋初值
- [ ] 使用析构函数释放对象所占空间
- [ ] 使用函数指针调用类成员
- [ ] 了解对象中的 this 指针
- [ ] 通过对象复制模拟菌类繁殖
- [ ] 实现运算符的重载





11.1　面向对象的编程思想

面向对象（Object Oriented）的英文缩写是 OO，它是一种设计思想，现在这种思想已经不单应用在软件设计上，数据库设计、计算机辅助设计（CAD）、网络结构设计、人工智能算法设计等领域都开始应用这种思想。

面向对象中的对象（Object）指的是客观世界中存在的对象，这个对象具有唯一性，对象之间各不相同，各有各的特点，每一个对象都有自动的运动规律和内部状态。对象与对象之间又是可以相互联系、相互作用的。概括地讲，面向对象技术是一种从组织结构上模拟客观世界的方法。

针对面向对象思想应用的不同领域，面向对象又可以分为面向对象分析（Object Oriented Analysis，OOA）、面向对象设计（Object Oriented Design，OOD）、面向对象编程（Object Oriented Programming，OOP）、面向对象测试（Object Oriented Test，OOT）和面向对象维护（Object Oriented Soft Maintenance，OOSM）。

客观世界中任何一个事物都可以看成一个对象，每个对象有属性和行为两个要素。属性就是对象的内部状态及自身的特点，行为就是改变自身状态的动作。

面向对象中的对象也可以是一个抽象的事物，可以从类似的事物中抽象出一个对象，例如圆形、正方形、三角形可以抽象得出的对象是简单图形，简单图形就是一个对象，它有自己的属性和行为，图形中边的个数是它的属性，图形的面积也是它的属性，输出图形的面积就是它的行为。

面向对象有 3 大特点，即封装、继承和多态。

（1）封装

封装有两个作用，一个是将不同的小对象封装成一个大对象，另一个是把一部分内部属性和功能对外界屏蔽。例如一辆汽车，它是一个大对象，它由发动机、底盘、车身和轮子等这些小对象组成。在设计时可以先对这些小对象进行设计，然后小对象之间通过相互联系确定各自大小等方面的属性，最后就可以安装成一辆汽车。

（2）继承

继承是和类密切相关的概念。继承性是子类自动共享父类数据结构和方法的机制，这是类之间的一种关系。在定义和实现一个类时，可以在一个已经存在的类的基础之上进行，把这个已经存在的类所定义的内容作为自己的内容，并加入若干新的内容。

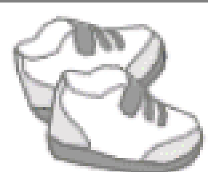
在类层次中，子类只继承一个父类的数据结构和方法，称为单重继承，子类继承了多个父类的数据结构和方法，则称为多重继承。

在软件开发中，类的继承性使所建立的软件具有开放性、可扩充性，这是信息组织与分类的行之有效的方法，它简化了对象、类的创建工作量，增加了代码的可重用性。

继承性是面向对象程序设计语言不同于其他语言的最重要的特点，是其他语言所没有的。采用继承性，使公共的特性能够共享，提高了软件的重用性。

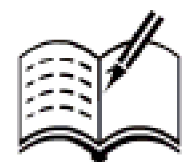
（3）多态

多态性是指相同的行为可作用于多种类型的对象上并获得不同的结果。不同的对象，收到同一消息可以产生不同的结果，这种现象称为多态性。多态性允许每个对象以适合自身的方式去响



应共同的消息。

11.1.1　面向过程



Note

过程编程的主要思想是先做什么后做什么，在一个过程中实现特定功能。一个大的实现过程还可以分成各个模块，各个模块可以按功能进行划分，然后组合在一起实现特定功能。在过程编程中，程序模块可以是一个函数，也可以是整个源文件。

过程编程主要以数据为中心，传统的面向过程的功能分解法属于结构化分析方法。分析者将对象系统的现实世界看作一个大的处理系统，然后将其分解为若干个子处理过程，解决系统的总体控制问题。在分析过程中，用数据描述各子处理过程之间的联系，整理各个子处理过程的执行顺序。

面向过程编程的一般流程为：现实世界→面向过程建模（流程图、变量、函数）→面向过程语言→执行求解。

过程编程的稳定性、可修改性和可重用性都比较差。

（1）软件重用性差

重用性是指同一事物不经修改或稍加修改就可多次重复使用的性质。软件重用性是软件工程追求的目标之一。处理不同的过程都有不同的结构，当过程改变时，结构也需要改变，前期开发的代码无法得到充分的再利用。

（2）软件可维护性差

软件工程强调软件的可维护性，强调文档资料的重要性，规定最终的软件产品应该由完整、一致的配置成分组成。在软件开发过程中，始终强调软件的可读性、可修改性和可测试性是软件重要的质量指标。面向过程编程由于软件的重用性差，造成维护时其费用和成本也很高，而且大量修改代码存在着许多未知的漏洞。

（3）开发出的软件不能满足用户需要

大型软件系统一般涉及各种不同领域的知识，面向过程编程往往描述软件的各个最低层的，针对不同领域设计不同的结构及处理机制，当用户需求发生变化时，就要修改最低层的结构。当处理用户需求变化较大时，过程编程将无法修改，可能导致软件的重新开发。

11.1.2　面向对象

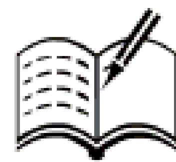
面向过程编程有费解的数据结构、复杂的组合逻辑、详细的过程和数据之间的关系、高深的算法，面向过程开发的程序可以描述成算法加数据结构。面向过程开发是分析过程与数据之间的边界在哪里，进而解决问题。面向对象则是从另一种角度思考，将编程思维设计成符合人的思维逻辑。

面向对象程序设计者的任务包括两个方面：一是设计所需的各种类和对象，即决定把哪些数据和操作封装在一起；二是考虑怎样向有关对象发送消息，以完成所需的任务。这时它如同一个总调度，不断地向各个对象发出消息，让这些对象活动起来（或者说激活这些对象），完成自己职责范围内的工作。



各个对象的操作完成了，整体任务也就完成了。显然，对一个大型任务来说，面向对象程序设计方法是十分有效的，它能大大降低程序设计人员的工作难度，减少出错机会。

面向对象开发的程序可以描述成“对象+消息”。面向对象编程的一般流程为：现实世界→面向对象建模（类图、对象、方法）→面向对象语言→执行求解。



Note

11.1.3　面向对象编程的特点

面向对象技术充分体现了分解、抽象、模块化、信息隐藏等思想，有效提高软件生产率、缩短软件开发时间、提高软件质量，是控制复杂度的有效途径。

面向对象不仅适合普通人员，也适合经理人员。降低维护开销的技术可以释放管理者的资源，将其投入到待处理的应用中。在经理们看来，面向对象不是纯技术的，它既能给企业的组织也能给经理的工作带来变化。

当一个企业采纳了面向对象，其组织将发生变化。类的重用需要类库和类库管理人员，每个程序员都要加入到两个组中的一个：一个是设计和编写新类组，另一个是应用类创建新应用程序组。面向对象不太强调编程，需求分析相对地将变得更加重要。

面向对象编程主要有代码容易修改、代码复用性高、满足用户需求 3 个特点。

（1）代码容易修改

面向对象编程的代码都是封装在类里面的，如果类的某个属性发生变化，只需要修改类中成员函数的实现即可，其他的程序函数不发生改变。如果类中属性变化较大，则使用继承的方法重新派生新类。

（2）代码复用性高

面向对象编程的类都是具有特定功能的封装，需要使用类中特定的功能，只需要声明该类并调用其成员函数即可。如果需要的功能在不同类，还可以进行多重继承，将不同类的成员封装到一个类中。功能的实现可以像积木一样随意组合，大大提高了代码的复用性。

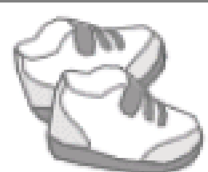
（3）满足用户需求

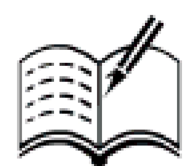
由于面向对象编程的代码复用性高，用户的要求发生变化时，只需要修改发生变化的类。如果用户的要求变化较大时，就对类进行重新组装，将变化大的类重新开发，功能没有发生变化的类可以直接拿来使用。面向对象编程可以及时地响应用户需求的变化。

11.2　类与对象

面向对象中的对象需要通过定义类来声明，对象一词是一种形象的说法，在编写代码过程中则是通过定义一个类来实现的。

C++类不同于汉语中的类、分类、类型，它是一个特殊的概念，可以是对同一类型事物进行抽象处理，也可以是一个层次结构中的不同层次节点。例如，将客观世界看成一个 object 类，动物是客观世界中的一小部分，定义为 Animal 类，狗是一种哺乳动物，是动物的一类，定义为 Dog 类，鱼也是一种动物，定义为 Fish 类，类的层次关系如图 11.1 所示。





Note

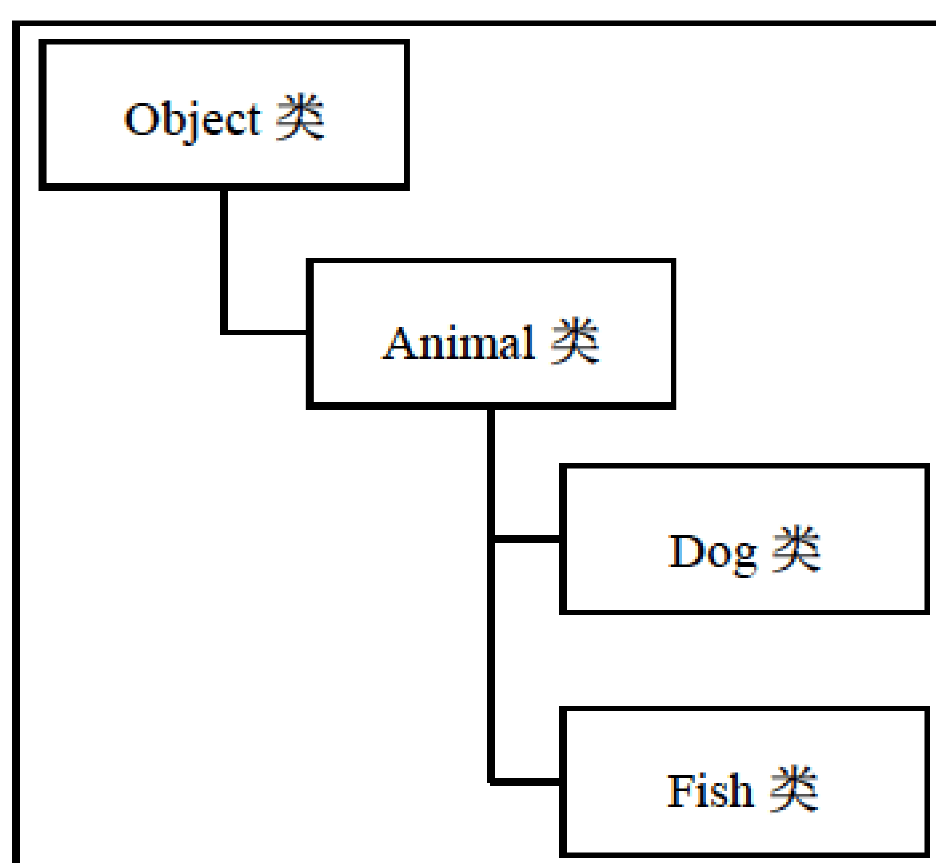


图 11.1　类的层次关系

类是一个新的数据类型，它和结构体有些相似，是由不同数据类型组成的集合体，但类比结构体增加了操作数据的行为，这个行为就是函数。

11.2.1　声明与定义类

前面已经对类的概念进行了说明，可以看出类是用户自己指定的类型。如果程序中要用到类这种类型，就必须自己根据需要进行声明，或者使用别人设计好的类。下面来看一下如何设计一个类。

类的声明格式如下：

```
class 类名标识符
{
[public:]
        [数据成员的声明]
        [成员函数的声明]
[private:]
        [数据成员的声明]
        [成员函数的声明]
[protected:]
        [数据成员的声明]
        [成员函数的声明]
};                              //注意这里需要加分号“;”
```

类的声明格式的说明如下：

☑ class 是定义类结构体的关键字，花括号内被称为类体或类空间。

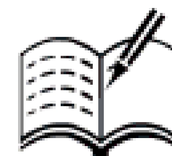
☑ 类名标识符指定的就是类名，类名就是一个新的数据类型，通过类名可以声明对象。

☑ 类的成员有函数和数据两种类型。

☑ 花括号内是定义和声明类成员的地方，关键字 public、private、protected 是类成员访问的修饰符。

类中的数据成员的类型可以是任意的，包含整型、浮点型、字符型、数组、指针和引用等，也可以是对象。另一个类的对象可以作为该类的成员，但是自身类的对象不可以作为该类的成员，而自身类的指针或引用又是可以作为该类的成员的。





例如，给出一个员工信息类声明：

```
class CPerson
{
	/*数据成员*/
	int m_iIndex;                        //声明数据成员
	char m_cName[25];                    //声明数据成员
	short m_shAge;                       //声明数据成员
	double m_dSalary;                    //声明数据成员
	/*成员函数*/
	short getAge();                      //声明成员函数
	int setAge(short sAge)               //声明成员函数
	int getIndex();                      //声明成员函数
	int setIndex(int iIndex);            //声明成员函数
	char* getName();                     //声明成员函数
	int setName(char cName[25]);         //声明成员函数
	double getSalary();                  //声明成员函数
	int setSalary(double dSalary);       //声明成员函数
};
```

在代码中，class 关键字是用来定义类这种类型的，CPerson 是定义的员工信息类名称，在大括号中包含了 4 个数据成员，分别表示 CPerson 类的属性，包含了 8 个成员函数表示 CPerson 类的行为。

11.2.2　在源文件中包含头文件

在前面的章节中会经常用到输入/输出流、字符串的头文件（.h），其中包含了数据和函数声明，而这些内容的实现部分一般会放到与头文件同名的实现源文件中（.cpp）。

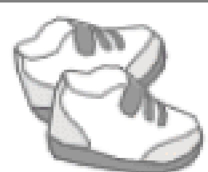
在一个源文件中使用#include 指令，可以将头文件的全部内容包含进来，也就是将另外的文件包含到本文件中。#include 指令使编译程序将另一源文件嵌入带有#include 的源文件，被读入的源文件必须用双引号或尖括号括起来。例如：

```
#include   "stdio.h"
#include   <stdio.h>
```

上面给出了双引号和尖括号的形式，这里说一下这两者之间的区别。用尖括号时，系统到存放 C++库函数头文件所在的目录中寻找要包含的文件，这种称为标准方式；用双引号时，系统先在用户当前目录中寻找要包含的文件，若找不到，再到存放 C++库函数头文件所在的目录中寻找要包含的文件。通常情况下，如果为了调用库函数用#include 命令来包含相关的头文件，则用尖括号，可以节省查找的时间。如果要包含的是用户自己编写的文件，一般用双引号，用户自己编写的文件通常是在当前目录中。如果文件不在当前目录中，双引号可给出文件路径。

11.2.3　实现一个类

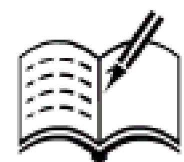
11.2.1 节只是在 CPerson 类中声明了类的成员，然而要使用这个类中的方法，即成员函数，



还要对其进行定义具体的操作。下面来看一下是如何定义类中的方法的。

第一种方法是将类的成员函数都定义在类体内。

以下代码都在 person.h 头文件内，类的成员函数都定义在类体内。



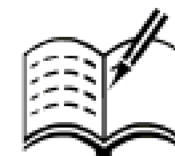
Note

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
class CPerson
{
public:
	//数据成员
	int m_iIndex;
	char m_cName[25];
	short m_shAge;
	double m_dSalary;
	//成员函数
	short getAge() { return m_shAge; }
	int setAge(short sAge)
	{
		m_shAge=sAge;
		return 0;                       //执行成功返回 0
	}
	int getIndex() { return m_iIndex; }
	int setIndex(int iIndex)
	{
		m_iIndex=iIndex;
		return 0;                       //执行成功返回 0
	}
	char* getName()
	{ return m_cName; }
	int setName(char cName[25])
	{
		strcpy(m_cName,cName);
		return 0;                       //执行成功返回 0
	}
	double getSalary() { return m_dSalary; }
	int setSalary(double dSalary)
	{
		m_dSalary=dSalary;
		return 0;                       //执行成功返回 0
	}
};
```

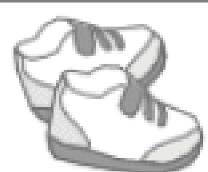
第二种方法是将类体内的成员函数的实现放在类体外，但如果类成员定义在类体外，需要用到域运算符“::”，放在类体内和类体外的效果是一样的。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```






```
class CPerson
{
public:
	//数据成员
	int m_iIndex;
	char m_cName[25];
	short m_shAge;
	double m_dSalary;
	//成员函数
	short getAge();
	int setAge(short sAge);
	int getIndex() ;
	int setIndex(int iIndex);
	char* getName() ;
	int setName(char cName[25]);
	double getSalary() ;
	int setSalary(double dSalary);
};
//类成员函数的实现部分
short CPerson::getAge()
{
	return m_shAge;
}
int CPerson::setAge(short sAge)
{
	m_shAge=sAge;
	return 0;                                   //执行成功返回 0
}
int CPerson::getIndex()
{
	return m_iIndex;
}
int CPerson::setIndex(int iIndex)
{
	m_iIndex=iIndex;
	return 0;                                   //执行成功返回 0
}
char* CPerson::getName()
{
	return m_cName;
}
int CPerson::setName(char cName[25])
{
	strcpy(m_cName,cName);
	return 0;                                   //执行成功返回 0
}
double CPerson::getSalary()
{
	return m_dSalary;
}
```



```
int CPerson::setSalary(double dSalary)
{
	m_dSalary=dSalary;
	return 0;	//执行成功返回 0
}
```



Note

前面两种方式都是将代码存储在同一个文件内。C++语言可以实现将函数的声明和函数的定义放在不同的文件内，一般在头文件中放入函数的声明，在实现文件中放入函数的实现文件。同样可以将类的定义放在头文件中，将类成员函数的实现放在实现文件内。存放类的头文件和实现文件最好和类名相同或相似。例如将 CPerson 类的声明部分放在 person.h 文件内，代码如下：

```
class CPerson
{
public:
	//数据成员
	int m_iIndex;
	char m_cName[25];
	short m_shAge;
	double m_dSalary;
	//成员函数
	short getAge();
	int setAge(short sAge);
	int getIndex() ;
	int setIndex(int iIndex);
	char* getName() ;
	int setName(char cName[25]);
	double getSalary() ;
	int setSalary(double dSalary);
};
```



说明：

代码中出现的关键字 public 表示成员变量可以被类外部调用，关于 public 等关键字所表示的访问权限将在第 12 章详细讲解。

将 CPerson 类的实现部分放在 person.cpp 文件内，代码如下：

```
#include "stdafx.h"
#include <iostream>
#include "person.h"
//类成员函数的实现部分
short CPerson::getAge()
{
	return m_shAge;
}
int CPerson::setAge(short sAge)
{
	m_shAge=sAge;
```





```
    return 0;                           //执行成功返回 0
}
int CPerson::getIndex()
{
    return m_iIndex;
}
int CPerson::setIndex(int iIndex)
{
    m_iIndex=iIndex;
    return 0;                           //执行成功返回 0
}
char* CPerson::getName()
{
    return m_cName;
}
int CPerson::setName(char cName[25])
{
    strcpy(m_cName,cName);
    return 0;                           //执行成功返回 0
}
double CPerson::getSalary()
{
    return m_dSalary;
}
int CPerson::setSalary(double dSalary)
{
    m_dSalary=dSalary;
    return 0;                           //执行成功返回 0
}
```

此时整个工程的所有文件如图 11.2 所示。

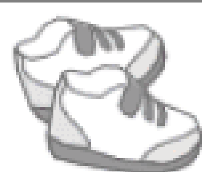


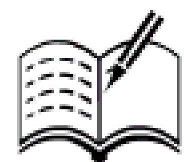
图 11.2　所有工程文件

关于类的实现有两点说明：

（1）类的数据成员需要初始化，成员函数还要添加实现代码。类的数据成员不可以在类的声明中初始化。例如下面的代码是不能通过编译的：

```
class CPerson
{
```





Note

```
        //数据成员
        int m_iIndex=1;                              //错误写法，不应该初始化的
        char m_cName[25]="Mary";                     //错误写法，不应该初始化的
        short m_shAge=22;                            //错误写法，不应该初始化的
        double m_dSalary=1700.00;                    //错误写法，不应该初始化的
        //成员函数
        short getAge();
        int setAge(short sAge)
        int getIndex();
        int setIndex(int iIndex);
        char* getName();
        int setName(char cName[25]);
        double getSalary();
        int setSalary(double dSalary);
};
```

（2）空类是 C++中最简单的类，其声明方式如下：

```
class CPerson{ };
```

空类只是起到占位的作用，需要时再定义类成员及实现。

11.2.4 实例化一个对象

对象也称为类的实例化。以上面的 person 类为例，定义了人所具有的属性、特征等，类内部的数据都是为描述每一个具体的人而准备的。这个具体的人指的就是类的实例化——对象。

定义一个新类后，就可以通过类名来声明一个对象。声明的形式如下：

```
类名 对象名表
Cperson jack;
```

声明多个对象如下：

```
CPerson LiMing,Tony;
```

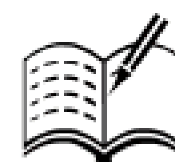
从程序的角度来说，对象 jack 与其他数据一样，具有类型 Cperson 这个自定义类型。从抽象上理解，jack 是定义的 Cperson 人类中的一个例子，一个典范。

11.2.5 访问类成员

访问类中的成员，形式如下：

```
Cperson jack;
jack.age = 25;                      //jack 的年龄为 25
jack.setName(“jack”);               //通过成员函数设定 jack 对象的 name 为 jack
```





【例 11.1】 通过实例化的对象访问类成员。

在本实例中，利用前文声明的类定义对象，然后使用该对象访问其中成员。

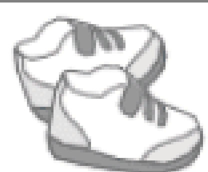
实例位置：光盘\MR\Instance\11\11.1

//person.h

```
class CPerson
{
public:
	//数据成员
	int m_iIndex;
	char m_cName[25];
	short m_shAge;
	double m_dSalary;
	//成员函数
	short getAge();
	int setAge(short sAge);
	int getIndex() ;
	int setIndex(int iIndex);
	char* getName() ;
	int setName(char cName[25]);
	double getSalary() ;
	int setSalary(double dSalary);
};
```

//person.cpp

```
#include "stdafx.h"
#include <iostream>
#include "person.h"
//类成员函数的实现部分
short CPerson::getAge()
{
	return m_shAge;
}
int CPerson::setAge(short sAge)
{
	m_shAge=sAge;
	return 0;                               //执行成功返回 0
}
int CPerson::getIndex()
{
	return m_iIndex;
}
int CPerson::setIndex(int iIndex)
{
	m_iIndex=iIndex;
	return 0;                               //执行成功返回 0
}
```





```
char* CPerson::getName()
{
	return m_cName;
}
int CPerson::setName(char cName[25])
{
	strcpy(m_cName,cName);
	return 0;				//执行成功返回 0
}
double CPerson::getSalary()
{
	return m_dSalary;
}
int CPerson::setSalary(double dSalary)
{
	m_dSalary=dSalary;
	return 0;				//执行成功返回 0
}
```

//main.cpp

```
#include"stdafx.h"
#include <iostream>
#include "Person.h"
using namespace std;
void main()
{
	int iResult=-1;
	CPerson p;
	iResult=p.setAge(25);
	if(iResult>=0)
		cout << "m_shAge is:" << p.getAge() << endl;

	iResult=p.setIndex(0);
	if(iResult>=0)
		cout << "m_iIndex is:" << p.getIndex() << endl;

	char bufTemp[]="Mary";
	iResult=p.setName(bufTemp);
	if(iResult>=0)
		cout << "m_cName is:" << p.getName() << endl;

	iResult=p.setSalary(1700.25);
	if(iResult>=0)
		cout << "m_dSalary is:" << p.getSalary() << endl;
}
```

p.setAge(25)引用类中的 setAge 成员函数，将参数中的数据赋值给数据成员，设置对象的属性。函数的返回值赋给 iResult 变量，通过 iResult 变量值判断函数 setAge 为数据成员赋值是否成功。如果成功再使用 p.getAge()得到赋值的数据，然后将其输出显示，如图 11.3 所示。



之后使用对象 p 依次引用成员函数 setIndex、setName 和 setSalary，然后通过对 iResult 变量的判断，决定是否引用成员函数 getIndex、getName 和 getSalary。

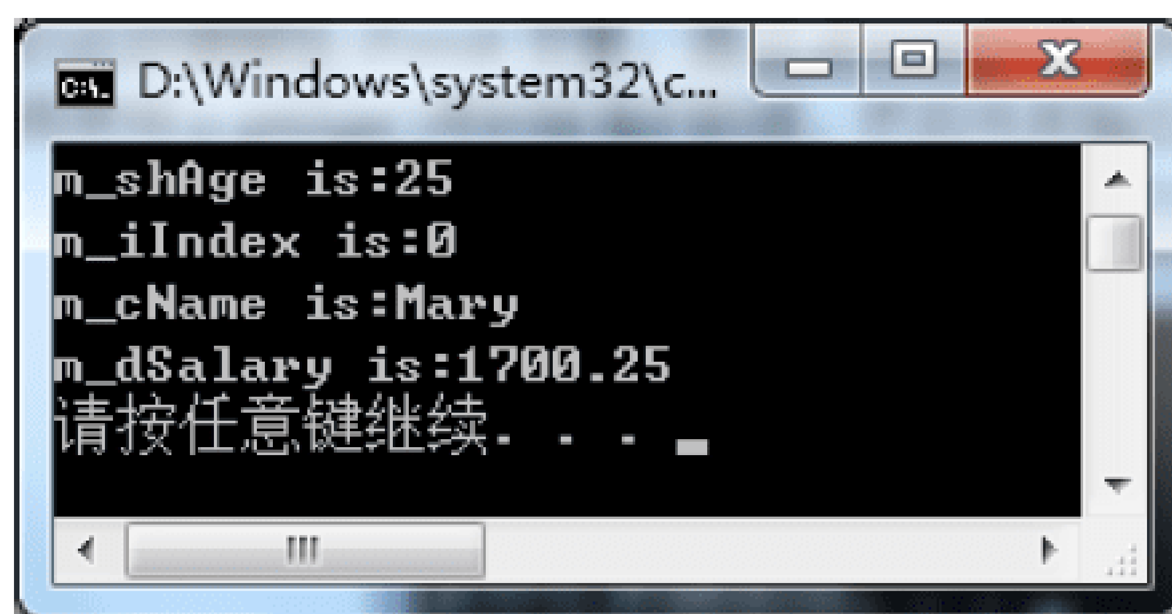


图 11.3　访问 CPerson 类成员



11.3　类的构造与析构

11.3.1　构造函数概述

在类的实例进入其作用域时，也就是建立一个对象，构造函数就会被调用，那么构造函数的作用是什么呢？当建立一个对象时，常常需要做某些初始化的工作，例如对数据成员进行赋值设置类的属性，而这些操作刚好放在构造函数中完成。前面使用了各种成员函数来初始化对象的属性，从代码的整洁度和效率来看都不是令人满意的。在构造函数中，可以完成初始化工作。

11.3.2　利用构造函数初始化成员变量

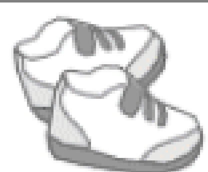
在类中声明一个和类同样名字的函数，以 CPerson 类为例：

```
class CPerson
{
    CPerson();
};
```

可以注意到的是，此函数的功能是在类构造时，完成初始化任务。此函数无返回值。实现此函数可以通过内部实现也可以通过外部实现，例如：

```
CPerson::CPerson()
{
    int m_iIndex=1;
    string m_sName="Mary";
    short m_shAge=22;
    double m_dSalary=1700.00;
}
```

构造函数中也可以具有参数，通过外部数据完成对象内部的初始化。



```
CPerson::CPerson(int index,string name,short age,double salary)
{
	int m_iIndex = index;
	string m_sName = name;
	short m_shAge = age;
	double m_dSalary =salary;
}
```



Note

当对象创建时，程序自动调用构造函数。同一个类中可以具备多个构造函数，通过这样的形式创建一个 CPerson 对象：

```
CPerson p1(0,"jack",22,7000);                //调用带参数的构造函数
CPerson p2 = CPerson(1, "tony",25,8000);     //调用带参数的构造函数
CPerson p;                                   //调用不带参数的构造函数
```

依照重载函数的特性，C++将找到对象创建时所调用的构造函数。

【例 11.2】 利用构造函数对成员变量赋初值。

实例位置：光盘\MR\Instance\11\11.2

Person.h 文件代码如下：

```
#include <string>
using std::string;
class CPerson
{
public:
	//构造函数
	CPerson(int index,string name,short age,double salary);
	CPerson();
	//数据成员
	int m_iIndex;
	string m_sName;
	short m_shAge;
	double m_dSalary;
	//成员函数
	short getAge();
	int setAge(short sAge);
	int getIndex() ;
	int setIndex(int iIndex);
	string getName() ;
	int setName(string sName);
	double getSalary() ;
	int setSalary(double dSalary);
};
short CPerson::getAge()
{
	return m_shAge;
}
int CPerson::getIndex()
{
```





```
	return m_iIndex;
}
```

main.cpp 文件代码如下：

```
#include "stdafx.h"
#include <iostream>
#include "Person.h"
using std::cout;
using std::endl;
void main()
{
	string str("tony");
	CPerson p1;
	CPerson p2 = CPerson(1, str,25,8000);
	cout <<"p1 的信息:"<<endl;
	cout << "m_shAge is:" << p1.getAge() << endl;
	cout << "m_iIndex is:" << p1.getIndex() << endl;
	cout << "m_cName is:" << p1.getName() << endl;
	cout << "m_dSalary is:" << p1.getSalary() << endl;
	cout <<"p2 的信息:"<<endl;
	cout << "m_shAge is:" << p2.getAge() << endl;
	cout << "m_iIndex is:" << p2.getIndex() << endl;
	cout << "m_cName is:" << p2.getName() << endl;
	cout << "m_dSalary is:" << p2.getSalary() << endl;
}
```

程序运行结果如图 11.4 所示。

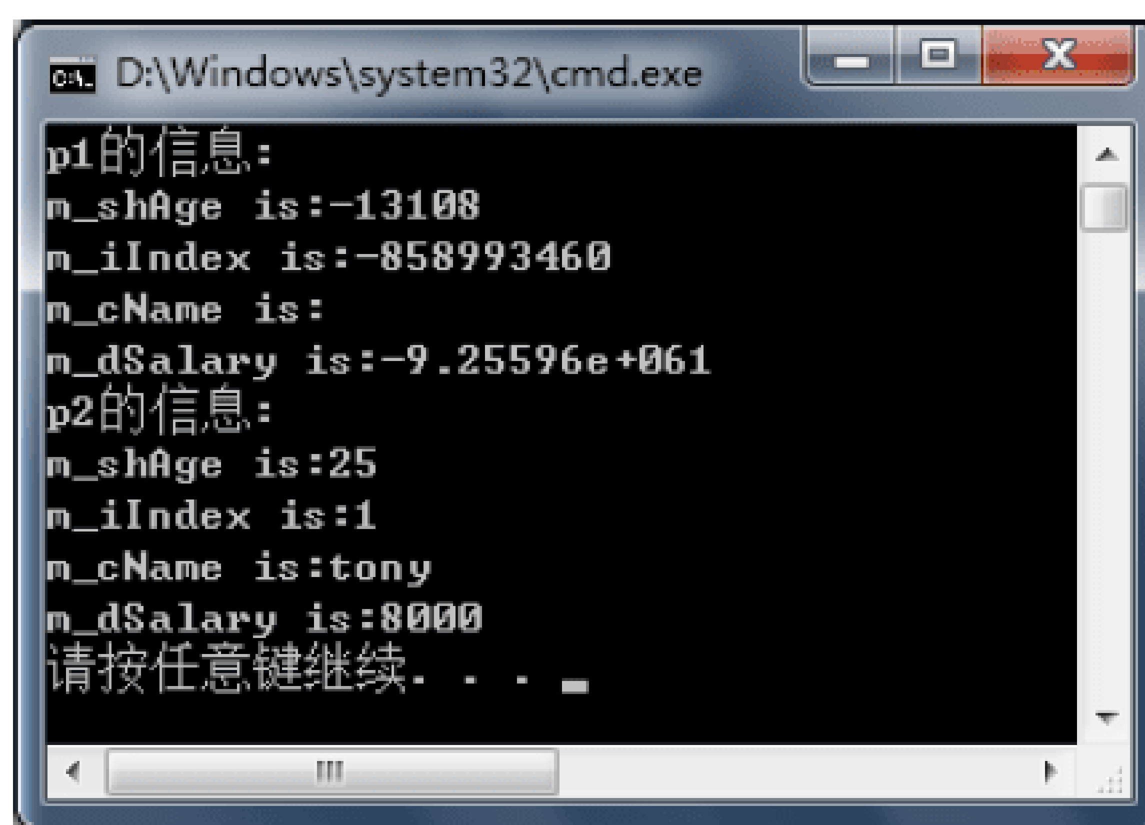
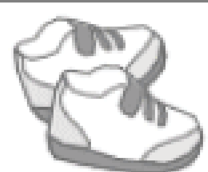


图 11.4　使用构造函数对象的初始化

在实例 11.2 中定义了两个构造函数，它们完成了各自的功能。实例 11.1 的代码中并没有定义构造函数，为什么不会出现问题？编译器执行过程中若发现类的代码中没有声明构造函数，则会分配给它一个没有内容的默认构造函数，例如：

```
CPerson ()
{
}
```

默认构造函数也称为无参构造函数。如同实例中的一样，默认构造函数可以被改写。若类中



没有声明无参构造函数，只声明了带参数的构造函数，此时称类没有默认构造函数。

使用构造函数初始化类成员的代码还允许以如下方式实现：

```
CPerson(int index,string name):m_index(intdex),m_name(name)
{
}
```



Note

它的作用是使用参数初始化自身的两个成员变量。



说明：

在实例 11.2 中，cpp 文件没有包含 string 的头文件也可使用 string 对象。这是因为 Person.h 中已经包含了 string .h，使用了 using 关键字。

11.3.3 析构一个类

当类的对象销毁时，编译器会调用类的析构函数。与构造函数相对的，析构函数主要执行的是收尾工作。构造函数名标识符和类名标识符相同，析构函数名标识符就是在类名标识符前面加～符号。

```
~CPerson();
```

下面将用一个实例来看看析构函数何时产生。

【例 11.3】 析构函数的调用。

实例位置：光盘\MR\Instance\11\11.3

title.h 文件中，声明了一个 title 类，代码如下：

```
#include <string>
#include <iostream>
using std::string;
class title{
public:
	title(string str);
	title();
	~title();
	string m_str;
};
```

title.cpp 文件中，实现了 title 类，代码如下：

```
#include "stdafx.h"
#include <iostream>
#include "title.h"
using std::cout;
using std::endl;
```





```
title::title(string str)
{
	m_str = str;
	cout<<str<<endl;
}
title::title()
{
	m_str = "无名标题";
	cout<<"这只是一个无名标题..."<<endl;
}
title::~title()
{
	cout<<"标题"<<m_str<<"要被销毁了"<<endl;
}
```

含有 main 函数的 cpp，程序的入口：

```
#include "stdafx.h"
#include "title.h"
#include <iostream>
using std::cout;
using std::endl;
int main()
{
	string str("Hello World!!!!");
	title out(str);
	if(true)
	{
		title t;
	}
	cout<<"if 执行完成"<<endl;
	return 0;
}
```

程序运行结果如图 11.5 所示。

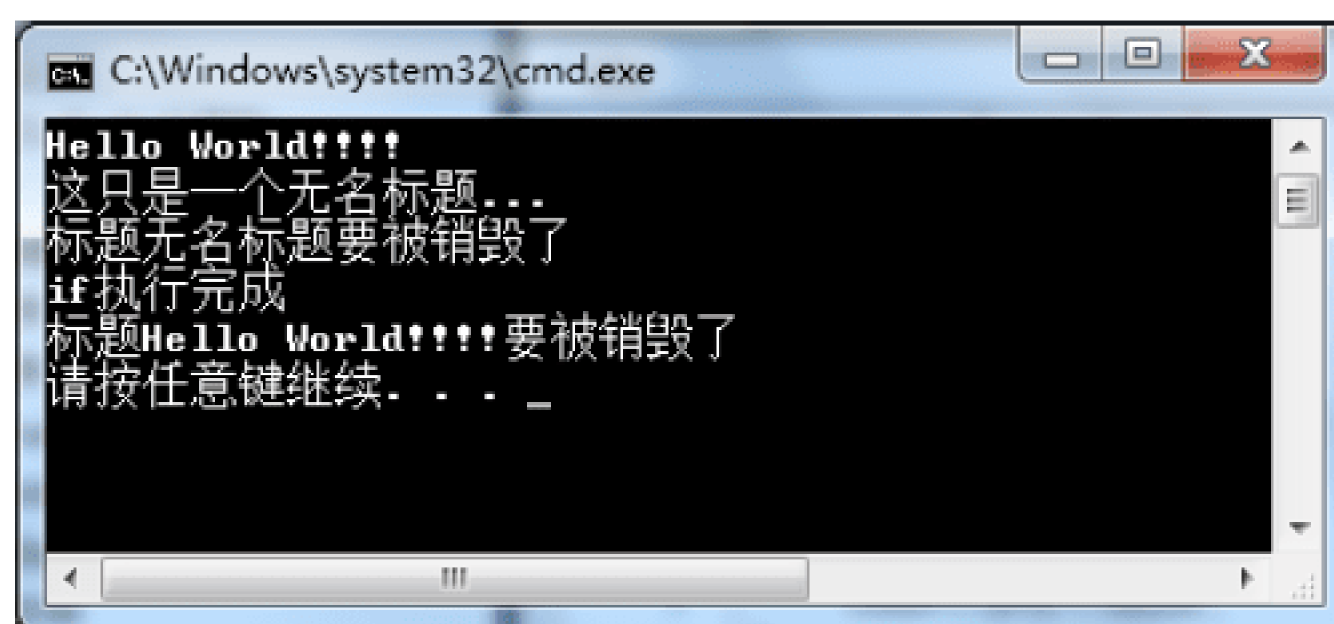
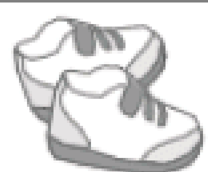
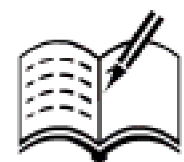


图 11.5　析构函数执行顺序

从执行结果来看，首先产生了 Hello World 标题，之后产生了 if 语句块中的无名标题。if 语句块执行完毕后，对象 t 销毁。之后用输出语句标识了分语句块的完成。程序执行完毕时，回收所有内存。out 标题销毁，自身析构函数被调用。





Note

11.4 定义静态成员

首先，让我们回顾一下静态数据的概念。静态数据在程序开始时即获得空间，直到程序结束后才会被回收。静态数据可以声明在函数体内，也可以声明在函数体外。那么，类可否有静态成员呢？答案是肯定的。

类中的静态成员与非静态成员有很大区别。从使用上来讲，调用静态成员不需要实例化对象，而是以如下形式调用：

```
类名::静态成员;
```

从设计类的思想来说，静态成员应该是类共用的。以人类作为例子，人们有很多的属性，如姓名、年龄、身高等。显然这些不是共用，一个具体人的名字，年龄不能交给所有人使用。在人类中添加一个“生存环境”属性。具体以何种数据类型来表示它视情况而定，如好坏（bool），或质量等级（int），但重要的是它一定是一个静态变量。因为对人类来讲，生存环境是共用的。类中的静态函数无法调用非静态成员变量，非静态成员变量在类未实例化时无法在内存中一直存在。若想在静态函数中使用某些成员变量，可以在形参列表中实例化本类的对象，这样在函数中可以使用该对象的成员。

【例 11.4】 我们共有一个地球。

实例位置：光盘\MR\Instance\11\11.4

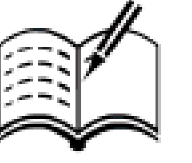
human.h 声明了 human 类：

```
#include <string>
using std::string;
class human
{
public:
	string m_name;
	human();
	human(string name);
	static int nTheEarth;
	static void GetFeel(human h);
	void Protect();
	void Destroy();
};
```

Human.cpp 实现了 human 类：

```
#include "stdafx.h"
#include "human.h"
#include <iostream>
using std::endl;
using std::cout;
```





```
int human::nTheEarth = 101;     //静态变量初始化!!!
human::human()
{
        m_name = "佚名";
}
human::human(string name)
{
        m_name = name;
}
void human::Destroy()
{
        human::nTheEarth-=20;
        cout<<m_name<<"破坏了环境"<<endl;
}
void human::Protect()
{
        human::nTheEarth+=6;
        cout<<m_name<<"尽微薄之力保护了环境"<<endl;
}
void human::GetFeel(human h)
{
        cout<<"环境现在的情况:";
        if(nTheEarth>100)
                cout<<"世界真美好"<<endl;
        else if(nTheEarth>80)
                cout<<"空气还算新鲜，但总觉得还是差了一些"<<endl;
        else if     (nTheEarth>60)
                cout<<"天不蓝，水不清，勉强可以忍受"<<endl;
        else if(nTheEarth>40)
                cout<<"树木稀少，黄沙漫天"<<endl;
        else if     (nTheEarth>20)
                cout<<"呼吸困难，水源稀缺"<<endl;
        else if     (nTheEarth>0)
                cout<<"难道世界末日到了么?"<<endl;
        if(nTheEarth<50)
        {
                cout<<"感到环境变的很糟糕,";
                h.Protect();
        }
}
```

程序入口：

```
#include "stdafx.h"
#include <iostream>
#include "human.h"
using std::cout;
using std::endl;
int main()
{
```


Note

```
        human h1("雷锋");
        human h2("某工厂老板");
        human h3("小明");
        human::GetFeel(h3);
        for(int day = 0;day<4;day++)
        {
              h1.Protect();
              h2.Destroy();
              if(day%2 == 0)
              h3.Destroy();
              else
              h3.Protect();
        }
        cout<<"现在的环境指数:"<<human::nTheEarth<<"，看来人类需要行动起来了..."<<endl;
        for(int day = 0;day<3;day++)
        {
              h1.Protect();
              human::GetFeel(h2);
              h3.Protect();
        }
        cout<<"现在的环境指数:"<<human::nTheEarth<<endl;
        return 0;
}
```

程序运行结果如图 11.6 所示。



图 11.6　保护环境人人有则

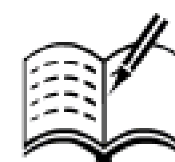
human 实现的 Protect 和 Destroy 方法都对静态成员 nTheEarth 进行了操作。可以看到的是每个 human 实例执行这两个函数后，静态成员 nTheEarth 都会变化，这个值是所有对象共用的。主函数使用区域符调用了 human 类的静态方法 GetFeel()。环境的变化，每个人看到的都是一样的，所以没有必要申请非静态成员函数来表示这一过程。





注意：

和其他静态变量一样，类的静态成员变量也需要初始化。初始化时必须在外部定义，而且要标明静态变量的类型。



Note

11.5 通过指针操作对象

指向相应类对象的指针，就是对象的指针。它的声明方法与其他类型一样：

```
类名* p;
```

类的指针可以调用它所指向对象的成员。形式如下：

```
p->类成员;
```

下面来看一个例子。

【例 11.5】 函数指针调用类成员。

实例位置：光盘\MR\Instance\11\11.5

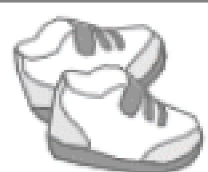
定义一个猫类，猫有名字，会发出叫声。

cat.h 文件代码如下：

```
#include <string>
using std::string;
class cat
{
public :
	string m_name;
	void sound();
	cat();
	cat(string name);
};
```

cat.cpp 文件代码如下：

```
#include "stdafx.h"
#include "cat.h"
#include <iostream>
using std::cout;
using std::endl;
cat::cat()
{
	m_name = "小猫";
}
cat::cat(string name)
```





Note

```
{
	m_name = name;
}
void cat::sound()
{
	cout<<"喵呜"<<endl;
}
```

程序入口 main.cpp 代码如下：

```
#include "stdafx.h"
#include <string>
#include <iostream>
#include "cat.h"
using std::cout;
using std::endl;
int main()
{
	cat c1 = cat("花花");
	cat* pC1 =&c1;
	cout<<"用手抚摸了"<<pC1->m_name<<endl;
	cout<<pC1->m_name<<"发出了叫声:"<<endl;
	pC1->sound();
}
```

程序运行结果如图 11.7 所示。

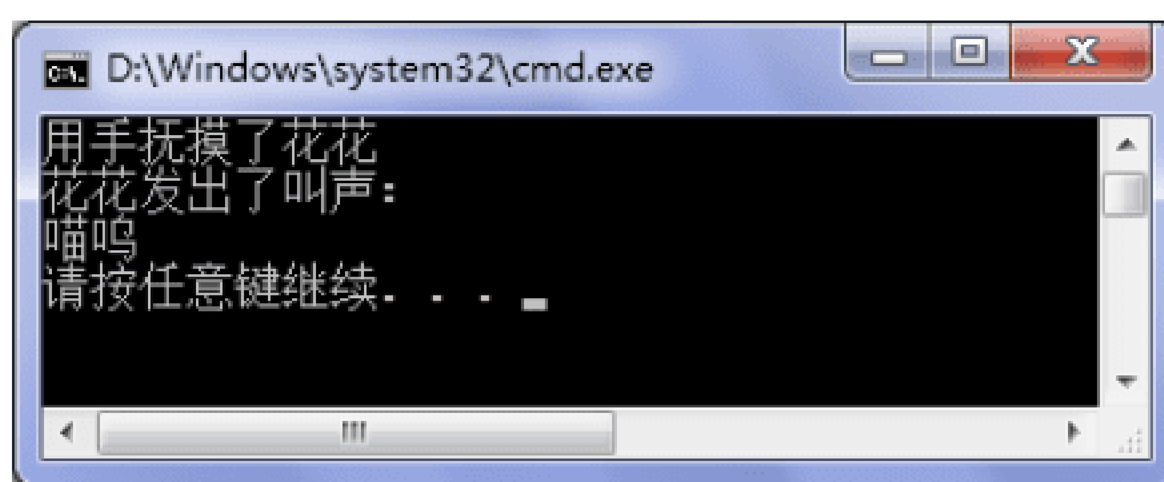


图 11.7　用指针调用成员

11.6　隐含的 this 指针

对于类的非静态成员，每一个对象都有自己的一份备份，即每个对象都有自己的数据成员，不过成员函数却是每个对象共享的。那么调用共享的成员函数是如何找到自己的数据成员的呢？答案就是通过类中隐藏的 this 指针。下面通过对例子的讲解来说明 this 指针的作用。

【例 11.6】 同一个类的不同对象数据。

实例位置：光盘\MR\Instance\11\11.6

```
class CBook                                        //定义一个 CBook 类
{
public:
```





```
        int m_Pages;                        //定义一个数据成员
        void OutputPages()                  //定义一个成员函数
        {
            cout<<m_Pages<<endl;            //输出信息
        }
};
int main(int argc, char* argv[])
{
        CBook vbBook,vcBook;                //定义两个 CBook 类对象
        vbBook.m_Pages = 512;               //设置 vbBook 对象的成员数据
        vcBook.m_Pages = 570;               //设置 vcBook 对象的成员数据
        vbBook.OutputPages();               //调用 OutputPages 方法输出 vbBook 对象的数据成员
        vcBook.OutputPages();               //调用 OutputPages 方法输出 vcBook 对象的数据成员
        return 0;
}
```

程序运行结果如图 11.8 所示。

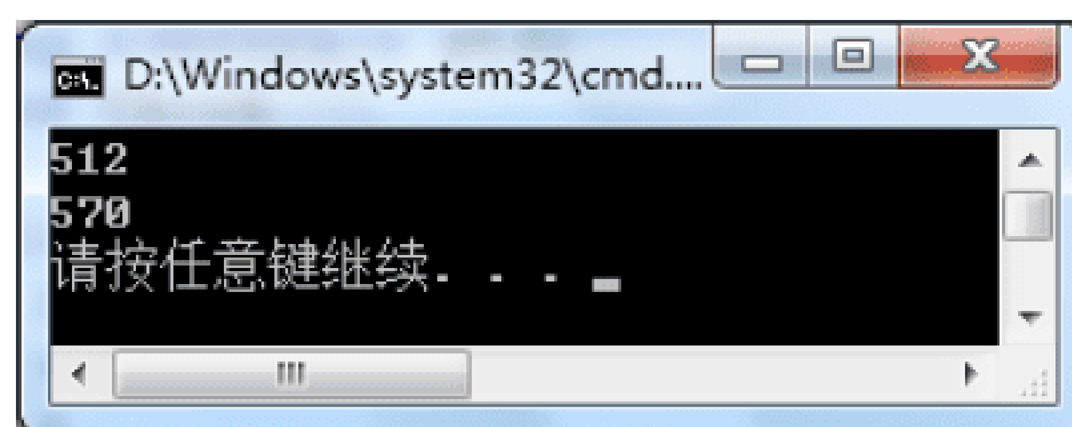


图 11.8　同一个类的不同对象数据

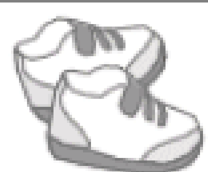
从图 11.8 中可以发现，vbBook 和 vcBook 两个对象均有自己的数据成员 m_Pages，在调用 OutputPages 成员函数时输出的均是自己的数据成员。在 OutputPages 成员函数中只是访问了 m_Pages 数据成员，每个对象在调用 OutputPages 方法时是如何区分自己的数据成员呢？答案是通过 this 指针。在每个类的成员函数（非静态成员函数）中都隐含包含一个 this 指针，指向被调用对象的指针，其类型为当前类类型的指针类型，在 const 方法中，为当前类类型的 const 指针类型。当 vbBook 对象调用 OutputPages 成员函数时，this 指针指向 vbBook 对象，当 vcBook 对象调用 OutputPages 成员函数时，this 指针指向 vcBook 对象。在 OutputPages 成员函数中，用户可以显式地使用 this 指针访问数据成员。例如：

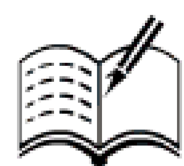
```
void OutputPages()
{
        cout <<this->m_Pages<<endl;             //使用 this 指针访问数据成员
}
```

实际上，编译器为了实现 this 指针，在成员函数中自动添加了 this 指针对数据成员的方法，类似于上面的 OutputPages 方法。此外，为了将 this 指针指向当前调用对象，并在成员函数中能够使用，在每个成员函数中都隐含包含一个 this 指针作为函数参数，并在函数调用时将对象自身的地址隐含作为实际参数传递。例如，以 OutputPages 成员函数为例，编译器将其定义为：

```
void OutputPages(CBook* this)                   //隐含添加 this 指针
{
        cout <<this->m_Pages<<endl;
}
```

在对象调用成员函数时，传递对象的地址到成员函数中。以“vc.OutputPages();”语句为例，编译器将其解释为“vbBook.OutputPages(&vbBook);”，这就使得 this 指针合法，并能够在成员函数中使用。





Note

11.7 复制对象

当函数以相应的类作为形参列表时，对象可以作为函数的参数传入。在学习函数时，我们曾提到过，值传递先复制实参产生副本。那么，对象的副本又会是什么样子的呢？

复制构造函数是指类的对象被复制时所调用的函数。下面两种情况对象都会调用复制构造函数。

☑ 将一个对象赋值给另外一个对象时

```
对象 1 = 对象 2;                //对象 1 与对象 2 所属类相同
对象 1(对象 2);
```

对象 2 的复制构造函数会被调用。

☑ 作为值传递的实参

```
funciton(对象 1);
```

在 function 函数体内，使用的是对象 1 的副本。所以之前会调用对象 1 的复制构造函数。

和构造函数一样，C++在未发现自定义的复制构造函数之前会创建一个默认的构造函数，其作用如上所示。

自定义的复制构造函数的声明格式为：

```
类名(类名& 形参)
```

值得注意的是，赋值构造函数是引用传递的函数。既然默认赋值构造函数已经完成复制工作，何时需要重新定义它呢？例如，一个类具有指针类型的数据，默认赋值构造函数执行之后，原对象和副本的指针成员指向的是同一片内存空间。通过指针改变该内存，就会改变两个对象实际应用的数据（也就是这块内存的内容）。这时可以自定义赋值构造函数，将两个指针的内存分离开。

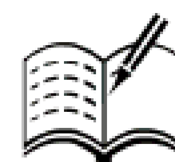
【例 11.7】 菌类的繁殖。

实例位置：光盘\MR\Instance\11\11.7

germ.h，声明了一个菌类：

```
#include <string>
using std::string;
class germ{
public:
        int m_age;
        string m_name;
        germ(germ& g);
        germ(string s);
        ~germ();
};
```





germ.cpp 实现了菌类：

```
#include "stdafx.h"
#include "germ.h"
#include <iostream>
using std::cout;
using std::endl;
germ::germ(string s)
{
	m_name = s;
	m_age = 1;
	cout<<"发现了"<<m_name<<endl;
}
germ::germ(germ& g)
{
	g.m_age +=1;
	this->m_age = 1;
	this->m_name =g.m_name + "的复制体";
	cout<<"产生了"<<g.m_name<<"的复制体"<<endl;
}
germ::~germ(){
	cout<<this->m_name<<"被消灭了"<<endl;
}
```

main.cpp 程序执行的入口：

```
#include "stdafx.h"
#include <iostream>
#include "germ.h"
using std::cout;
using std::endl;
germ copyGerm(germ gc)
{
	return gc;
}
int main()
{
	germ g1("有氧菌");
	germ g2(g1);
	germ g3("无氧菌");
	germ g4 = g3;
	germ g5 = copyGerm(g4);
	return 0;
}
```

程序运行结果如图 11.9 所示。

从执行结果来分析一下代码。首先在主函数中产生了 g1 对象，由复制构造函数产生了 g1 的





赋值体 g2——有氧菌复制体。之后定义了 g3——无氧菌。通过复制构造函数产生了 g3 的复制体 g4。前 4 行输出即是上面所述的过程。g5 的产生前，g5 所在的赋值语句等号右边的 copyGerm 函数调用，传递的实参为 g4——无氧菌的复制体。像本节开始提到的那样，值传递实参对象产生副本，副本就是形参 gc——无氧菌复制品的复制体。函数执行完毕后，传递回临时变量，内容是 gc。g5 的值经过了赋值语句所以它是 gc（临时变量使用的内存）的复制品。

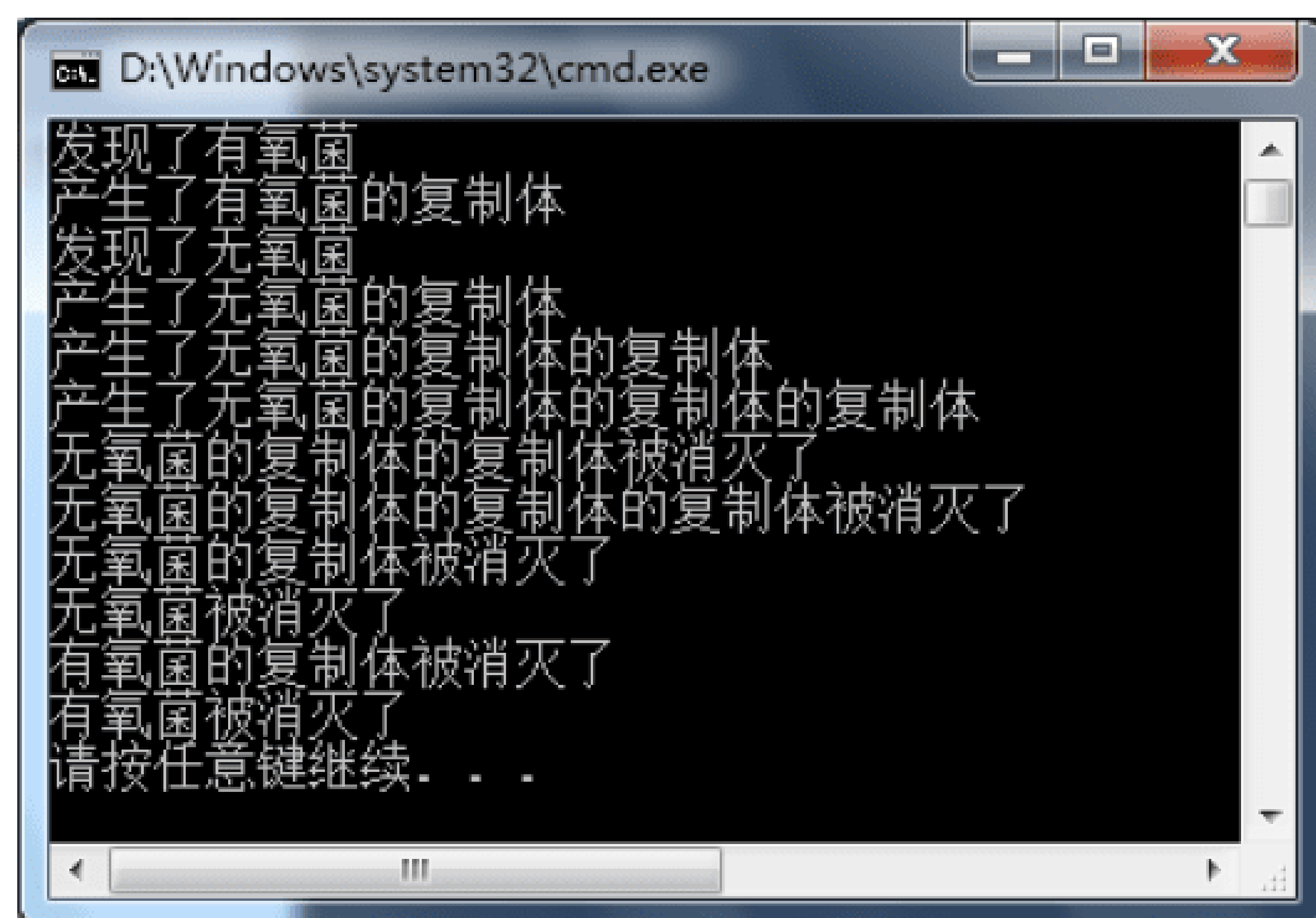
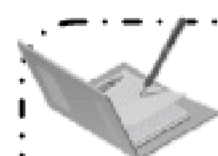


图 11.9　复制过程



说明：

通过本例读者可以知道何时 C++会调用类的复制构造函数。临时变量的应用在右值的概念章节中曾提到过。在这个事例中可以清楚地看到，临时变量是 C++提供的一个暂时的内存存储，用途只是赋值给其他数据。

11.8　声明 const 对象

当一个对象创建之后，不希望它的任何数据发生改变，则可以将其直接声明为 const 对象：

```
const 类名 对象名
```

值得注意的是，const 对象必须初始化。依照之前的一贯说法，这是一个只读对象。我们可以调用它的数据和函数，但是不可以对它们进行修改。除此之外，const 对象的 this 指针也是常量。在 11.6 节曾经提到过，成员函数在自己的函数体内自动为成员变量加上了 this 指针。如何使这些内存指针转换为 const 呢？仍然需要 const 关键字，函数声明形式如下：

```
返回类型 函数名(参数列表)   const;
```

即在函数头结尾加上 const。只能对类中的函数做如此声明，对外部的函数无效。

下面用一个实例进一步说明 const 对象的使用方法。





【例 11.8】 标准尺寸。

实例位置：光盘\MR\Instance\11\11.8

box.h 代码如下：

```
class box
{
public:
	int m_lenth;			//长
	int m_width;			//宽
	int m_hight;			//高
	box(int lenth,int width,int hight);
	bool Compare(box b) const;
};
```

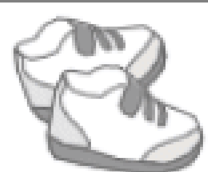
box.cpp 代码如下：

```
#include "stdafx.h"
#include <iostream>
#include "box.h"
using std::cout;
using std::endl;

box::box(int lenth,int width,int hight)
{
	m_lenth = lenth;
	m_width = width;
	m_hight = hight;
	cout<<"刚刚制作的盒子长:"<<lenth<<"宽:"<<width<<"高:"<<hight<<endl;
}
bool box::Compare(box b) const
{
	return (m_lenth == b.m_lenth)&(m_width == b.m_width)&(m_hight == b.m_hight);
}
```

main.cpp 程序入口：

```
#include "stdafx.h"
#include "box.h"
#include <iostream>
using std::cout;
using std::endl;
using std::cin;
int main()
{
	const box styleBox(4,2,3);
	cout<<"标准盒子创建完成"<<endl;
	box temp(1,1,1);
	while(styleBox.Compare(temp) != true)
	{
```






```
            cout<<"刚才的盒子不合适"<<endl;
            int lenth;
            int width;
            int hight;
            cout<<"请输入新盒子的数据，使它符合标准盒子的大小"<<endl;
            cin>>lenth;
            cin>>width;
            cin>>hight;
            temp = box(lenth,width,hight);
        }
        cout<<"盒子刚好合适，恭喜你"<<endl;
        return 0;
}
```

程序运行结果如图 11.10 所示。

假如试图改变 sytleBox 的长、宽、高，编译器就会报错，如图 11.11 所示。

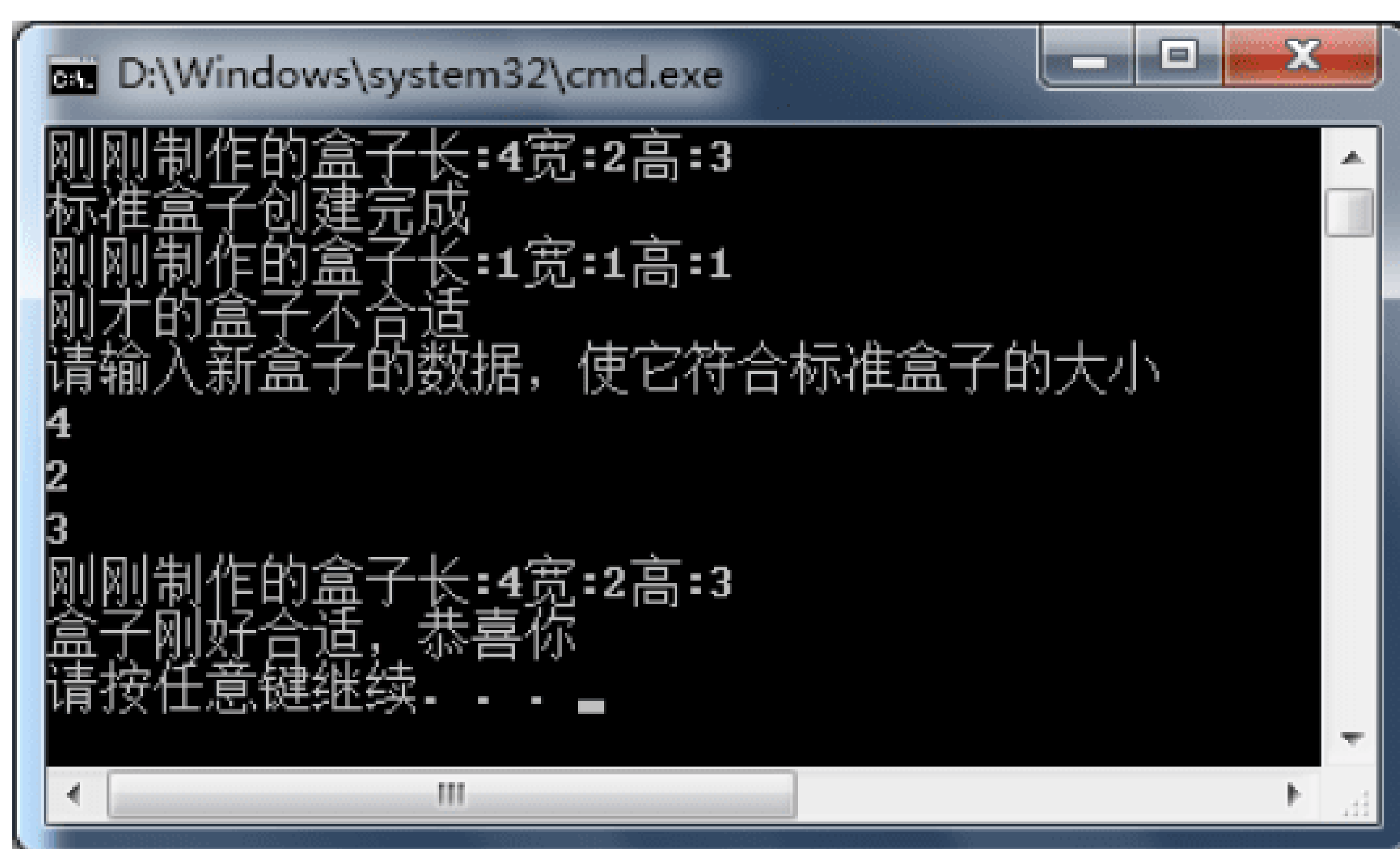


图 11.10　标准尺寸



图 11.11　const 对象的成员变量不可修改

11.9　申请对象数组

在数组一章中我们曾了解到，数组是通过指针分配到的一段额定大小的内存空间。同样地，数组也可以包含对象。声明对象数组的形式如下：

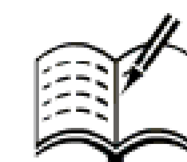
```
box boxArray[5];
box boxArray2[2] = {box(1,1,1),box(2,2,2)};
box boxArray3[3] = (3,styleBox);
```

值得注意的是，第一种申请对象数组的方法必须保证类中含有默认的构造函数，否则编译器将会报错。同样地，可以通过对象指针申请动态对象数组。

```
box* pbox;
pbox = new box[n];          //n 为整数
```

同时需要确认 box 中含有默认（无参）构造函数。





【例 11.9】 批量化生产。

实例位置：光盘\MR\Instance\11\11.9

box.h

```
class box{
public:
	//类成员变量
	float m_lenth;                                          //长
	float m_width;                                          //宽
	float m_hight;                                          //高
	int Number;                                             //流水线编号
	//类成员函数
	box(float lenth,float width,float hight);
	box();
	bool Compare(const box b) const;
	void ToCheck();                                         //显示当前盒子的规格
	void Rebuild(float lenth,float width,float hight);      //重新定义长、宽、高
};
```

box.cpp

```
#include "stdafx.h"
#include <iostream>
#include "box.h"
using std::cout;
using std::endl;
box::box()
{
	m_lenth = 1.000f;
	m_width = 1.000f;
	m_hight = 1.000f;
	cout<<"制作的盒子长:"<<m_lenth<<"宽:"<<m_width<<"高:"<<m_hight<<endl;
}
box::box(float lenth,float width,float hight)
{
	m_lenth = lenth;
	m_width = width;
	m_hight = hight;
	cout<<"定制的盒子长:"<<lenth<<"宽:"<<width<<"高:"<<hight<<endl;
}
bool box::Compare(const box b) const
{
	return (m_lenth == b.m_lenth)&(m_width == b.m_width)&(m_hight == b.m_hight);
}
void box::ToCheck()
{
	cout<<"本盒子现在长:"<<m_lenth<<"宽:"<<m_width<<"高:"<<m_hight<<endl;
}
void box::Rebuild(float lenth,float width,float hight)
```



```
{
	m_lenth = lenth;
	m_width = width;
	m_hight = hight;
}
```



Note

程序入口 main.cpp：

```
#include "stdafx.h"
#include "box.h"
#include <iostream>
using std::cout;
using std::endl;
using std::cin;
bool check(float a,float b,float c)
{
	return (a>0)&(b>0)&(c>0)&(a<100)&(b<100)&(c<100);
}
int main()
{
	float lenth;
	float hight;
	float width;
	cout<<"请输入您所需要盒子，长、宽、高"<<endl;
	while(cin>>lenth,cin>>hight,cin>>width,!check(lenth,width,hight))
	{
		cout<<"抱歉，你所输入的规格超出我们的制作水平，请重新输入"<<endl;
	}
	const box styleBox(lenth,width,hight);
	cout<<"请输入您的订单个数:"<<endl;
	int count;
	while(cin>>count,!((count>0)&(count<6)))			//数字检查
	{
		if(count>5)
		{
			cout<<"抱歉，订单数额超出生产水平，请重新输入"<<endl;
		}
		else{
			cout<<"请确认输入的数值为正数，请重新输入"<<endl;
		}
	}
	box* boxArray ;
	boxArray = new box[count];					//动态对象数组
	bool bOk = false;
	for(int i=0;i<count;i++)
	{
		boxArray[i].Rebuild(lenth,width,hight);
		boxArray[i].ToCheck();
		if(styleBox.Compare(boxArray[i]))
		{
```



```
                cout<<"此产品符合规格"<<endl;
            }
        }
        delete []boxArray;
        return 0;
}
```



Note

程序中将长、宽、高定义为了浮点数类型，如果输入超过精度，则会将超过精度的最后一位按照四舍五入的方式进位，程序运行结果如图 11.12 所示。

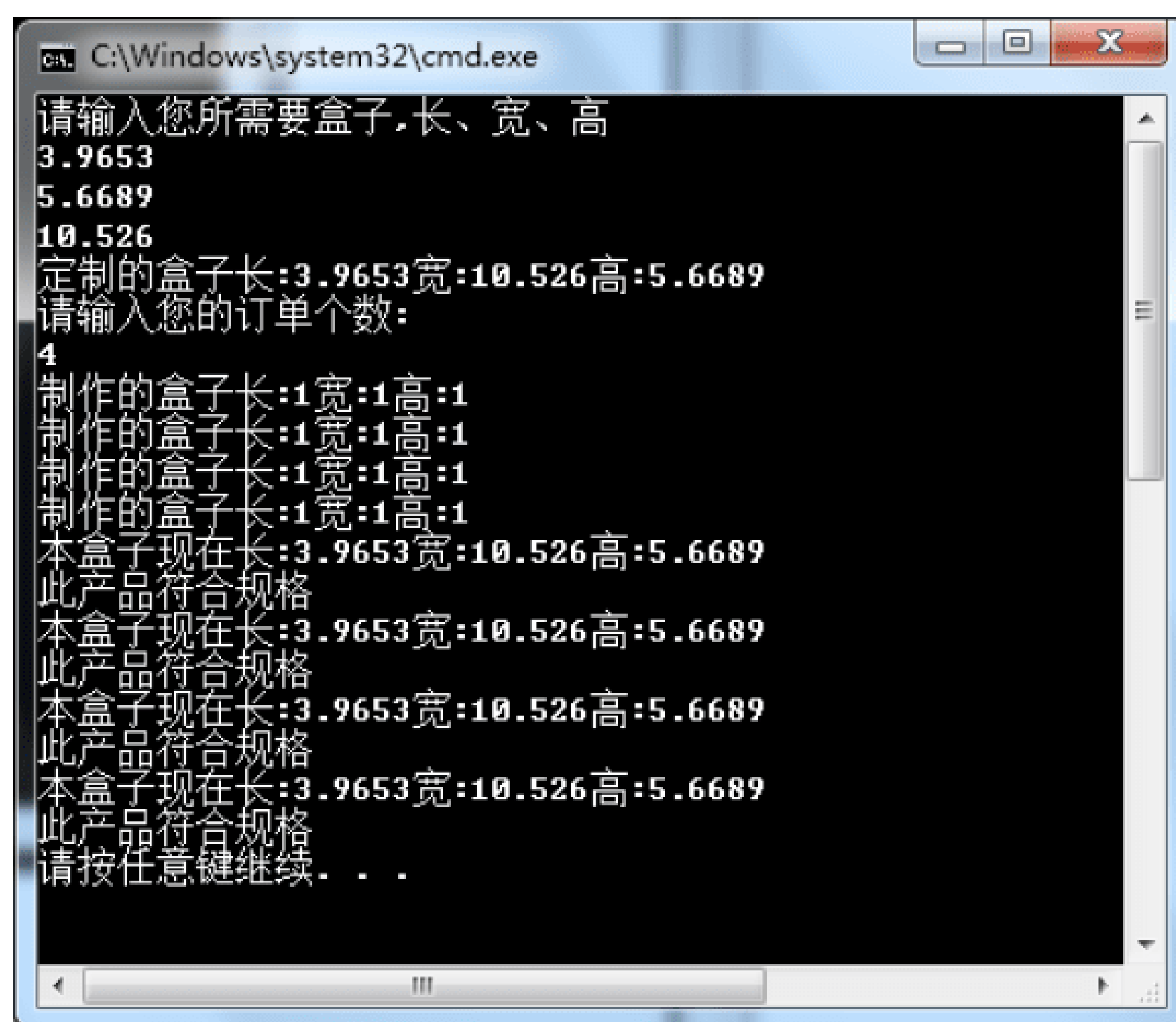


图 11.12　执行结果

11.10　C++中的友元

11.10.1　友元机制

在讲述类的内容时说明了隐藏数据成员的好处，但是有些时候，类会允许一些特殊的函数直接读写其私有数据成员。

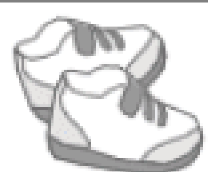
使用 friend 关键字可以让特定的函数或者别的类的所有成员函数对私有数据成员进行读写。这既可以保持数据的私有性，又能够使特定的类或函数直接访问私有数据。

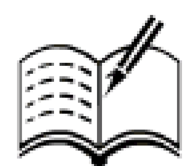
有时候，普通函数需要直接访问一个类的保护或私有数据成员。如果没有友元机制，则只能将类的数据成员声明为公共的，从而任何函数都可以无约束地访问它。

普通函数需要直接访问类的保护或私有数据成员的原因主要是为了提高效率。

例如，没有使用友元函数的情况如下：

```
#include <iostream.h>
class CRectangle
```





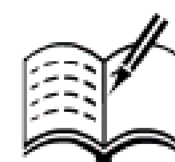

```
{
public:
        CRectangle()
        {
                m_iHeight=0;
                m_iWidth=0;
        }
        CRectangle(int iLeftTop_x,int iLeftTop_y,int iRightBottom_x,int iRightBottom_y)
        {
                m_iHeight=iRightBottom_y-iLeftTop_y;
                m_iWidth=iRightBottom_x-iLeftTop_x;
        }
        int getHeight()
        {
                return m_iHeight;
        }
        int getWidth()
        {
                return m_iWidth;
        }
protected:
        int m_iHeight;
        int m_iWidth;
};
int ComputerRectArea(CRectangle & myRect)                    //不是友元函数的定义
{
        return myRect.getHeight()*myRect.getWidth();
}
void main()
{
        CRectangle rg(0,0,100,100);
        cout << "Result of ComputerRectArea is :"<< ComputerRectArea(rg) << endl;
}
```

在代码中可以看到，ComputerRectArea 函数在定义时只能对类中的函数进行引用，因为类中的函数属性都为公有属性，对外是可见的，但是数据成员的属性为受保护属性，对外是不可见的，所以只能使用公有成员函数得到想要的值。

下面来看一下使用友元函数的情况：

```
#include <iostream.h>
class CRectangle
{
public:
        CRectangle()
        {
                m_iHeight=0;
                m_iWidth=0;
        }
        CRectangle(int iLeftTop_x,int iLeftTop_y,int iRightBottom_x,int iRightBottom_y)
```





```
	{
		m_iHeight=iRightBottom_y-iLeftTop_y;
		m_iWidth=iRightBottom_x-iLeftTop_x;
	}
	int getHeight()
	{
		return m_iHeight;
	}
	int getWidth()
	{
		return m_iWidth;
	}
	friend int ComputerRectArea(CRectangle & myRect);	//声明为友元函数
protected:
	int m_iHeight;
	int m_iWidth;
};
int ComputerRectArea(CRectangle & myRect)			//友元函数的定义
{
	return myRect.m_iHeight*myRect.m_iWidth;
}
void main()
{
	CRectangle rg(0,0,100,100);
	cout << "Result of ComputerRectArea is :"<< ComputerRectArea(rg) << endl;
}
```

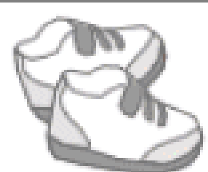
在 ComputerRectArea 函数的定义中可以看到使用 CRectangle 的对象可以直接引用其中的数据成员，这是因为在 CRectangle 类中将 ComputerRectArea 函数声明为友元了。

从中可以看到使用友元保持了 CRectangle 类中数据的私有性，起到了隐藏数据成员的好处，又使得特定的类或函数可以直接访问这些隐藏数据成员。

11.10.2　定义友元类

对于类的私有方法，只有在该类中允许访问，其他类是不能访问的。但在开发程序时，如果两个类的耦合度比较紧密，能够在一个类中访问另一个类的私有成员会带来很大的方便。C++语言提供了友元类和友元方法（或者称为友元函数）来实现访问其他类的私有成员。当用户希望另一个类能够访问当前类的私有成员时，可以在当前类中将另一个类作为自己的友元类，这样在另一个类中就可以访问当前类的私有成员了。例如，定义友元类：

```
class CItem						//定义一个 CItem 类
{
private:
	char m_Name[128];				//定义私有的数据成员
	void OutputName()				//定义私有的成员函数
	{
```





```
            printf("%s\n",m_Name);                          //输出 m_Name
        }
public:
        friend  class  CList;                               //将 CList 类作为自己的友元类
        void SetItemName(const char* pchData)               //定义公有成员函数，设置 m_Name 成员
        {
            if (pchData != NULL)                            //判断指针是否为空
            {
                strcpy(m_Name,pchData);                     //赋值字符串
            }
        }
        CItem()                                             //构造函数
        {
            memset(m_Name,0,128);                           //初始化数据成员 m_Name
        }
};
class CList                                                 //定义类 CList
{
private:
        CItem m_Item;                                       //定义私有的数据成员 m_Item
public:
        void OutputItem();                                  //定义公有成员函数
};
void CList::OutputItem()                                    //OutputItem 函数的实现代码
{
        m_Item.SetItemName("BeiJing");                      //调用 CItem 类的公有方法
        m_Item.OutputName();                                //调用 CItem 类的私有方法
}
```

在定义 CItem 类时，使用 friend 关键字将 CList 类定义为 CItem 类的友元，这样 CList 类中的所有方法都可以访问 CItem 类中的私有成员了。在 CList 类的 OutputItem 方法中，语句“m_Item.OutputName()”演示了调用 CItem 类的私有方法 OutputName。

11.11 重载运算符

11.11.1 重载算术运算符

在字符串的章节中曾介绍过 string 类型的数据，它是 C++标准模板库提供给编程者的一个类。string 类支持使用加号+连接两个 string 对象。但是使用两个 string 对象相减却是非法的，其中的原理就是 C++所提供类中重载运算符的功能。在 string 类中定义了运算符+和+=两个符号的使用方法，这种使用方法实质上是一种成员函数。

关键字 operator 是专门实现运算符重载的关键字。在类成员中，定义一个这样形式的函数：

```
返回值类型  operator  重载的运算符(参数列表)
```





【例 11.10】 重载加号运算符。

实例位置：光盘\MR\Instance\11\11.10

```
#include "stdafx.h"
#include "box.h"
#include <iostream>
using namespace std;
class CBook{
public:
	CBook(int iPage)
	{
		m_iPage = iPage;
	}
	CBook operator+(CBook b)
	{
		return CBook(m_iPage+b.m_iPage);
	}
	void display()
	{
		cout<<m_iPage<<endl;
	}
protected:
	int m_iPage;
};
void main()
{
	CBook bk1(10);
	CBook bk2(20);
	CBook tmp(0);
	tmp = bk1+bk2;
	tmp.display();
}
```

程序运行结果如图 11.13 所示。

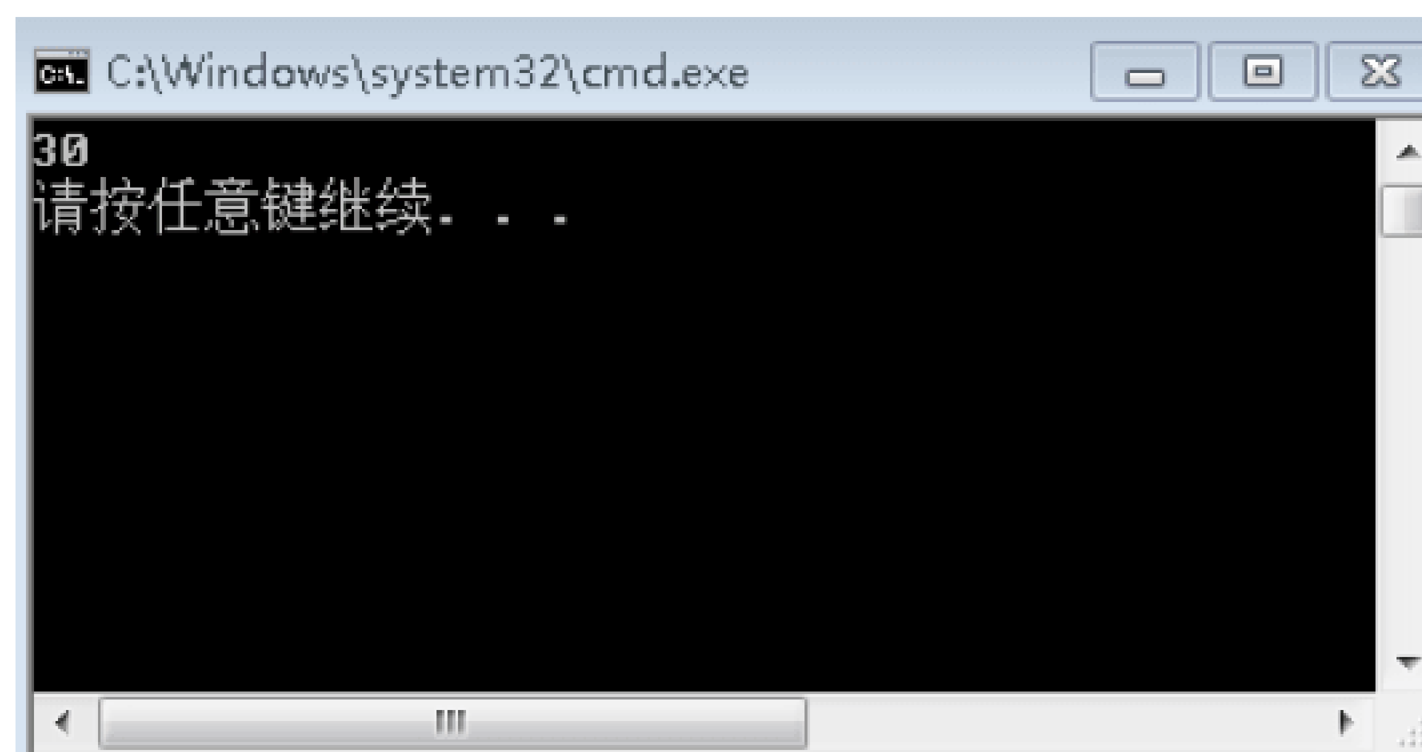
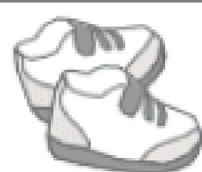


图 11.13　两个盒子“相加”

类 Cbook 重载了求和运算符之后，由它声明了两个对象 bk1 和 bk2，可以像两个整型变量一样相加。



11.11.2　重载比较运算符



Note

除了加减乘除外，比较运算符也可以被用作重载。根据比较运算符的运算规则，在重载它们时最好贴近它们的定义。

【例 11.11】 重载比较运算符。

实例位置：光盘\MR\Instance\11\11.11

box.h 代码如下：

```
class box{
public:
	//类成员变量
	float m_lenth;                          //长
	float m_width;                          //宽
	float m_hight;                          //高
	//类成员函数
	box(float lenth,float width,float hight);
	bool operator >(const box b) const;     //重载>
	bool operator <(const box b) const;     //重载<
};
```

在 box.cpp 中添加实现部分：

```
#include "StdAfx.h"
#include "box.h"
#include <iostream>
using namespace std;
bool box::operator> (const box b) const
{
	return (m_lenth > b.m_lenth)&&(m_width > b.m_width)&&(m_hight > b.m_hight);
}
bool box::operator<(const box b) const
{
	return (m_lenth < b.m_lenth)&&(m_width < b.m_width)&&(m_hight < b.m_hight);
}
box::box(float lenth,float width,float hight)
{
	m_lenth = lenth;
	m_width = width;
	m_hight = hight;
	cout<<"定制的盒子长："<<m_lenth<<"宽："<<m_width<<"高："<<m_hight<<endl;
}
```

程序入口 main.cpp 代码如下：

```
#include "stdafx.h"
#include "box.h"
#include <iostream>
```





```
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
        box boxA(4.44f,3.33f,5.55f);
        box boxB(14.44f,13.33f,15.55f);
        box boxC(24.44f,3.33f,1.55f);
        if(boxA>boxB)
        {
                cout<<"盒子 A 能完全容纳下盒子 B"<<endl; //这里容纳指的是按常规摆放时的容纳
        }
        else if(boxA<boxB)
        {
                cout<<"盒子 B 能完全容纳下盒子 A"<<endl;
        }
        else
        {
                cout<<"这两个盒子不能相互容纳"<<endl;
        }
        if(boxC>boxB)
        {
                cout<<"盒子 C 能完全容纳下盒子 B"<<endl; //这里容纳指的是按常规摆放时的容纳
        }
        else if(boxC<boxB)
        {
                cout<<"盒子 B 能完全容纳下盒子 C"<<endl;
        }
        else
        {
                cout<<"这两个盒子不能相互容纳"<<endl;
        }
        return 0;
}
```

程序运行结果如图 11.14 所示。

除算术运算符、比较运算符之外，逻辑运算符、位运算符、赋值运算符（=，+=）、调用运算符，即( )等都可以被重载。赋值运算符被重载后失去原来的定义，转为重载运算符函数。在重载运算符时，最好将运算符应用于参数、成员变量等，使运算符的意义与重载的意义相近。

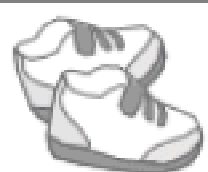


图 11.14　执行结果

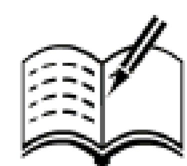
11.12　综合应用

11.12.1　用户与留言

【例 11.12】 本例将设计一个具有用户名、密码、年龄、性别共 4 项成员的用户类，它能



够以记名的形式在控制台中留言。在设计这个用户类时，首先应声明它的成员变量与成员函数。它的成员变量就是名字、密码、年龄和性别 4 个属性。实例化用户类时，自身的属性应该被初始化。在类中添加一个留言的函数，在函数中输出名字成员变量与留言内容。参考代码如下：



Note

实例位置：光盘\MR\Instance\11\11.12

//user.h

```
#include <string>
using std::string;
class user
{
public:
	user(string name,string password,int age,string sex);
	string m_name;                          //用户名
	string m_password;                      //密码
	int m_age;                              //年龄
	string m_sex;
	void LeaveMessage(string s);
};
```

//user.cpp

```
#include "stdafx.h"
#include "user.h"
#include <iostream>
using std::cout;
using std::endl;
user::user(string name,string password,int age,string sex)
{
	this->m_age = age;
	this->m_name = name;
	this->m_password = password;
	this->m_sex = sex;
}
void user::LeaveMessage(string s)
{
	std::cout<<m_name<<"..:"<<s<<endl;
}
```

//main.cpp

```
#include "stdafx.h"
#include "user.h"
int main(int argc, _TCHAR* argv[])
{
	user man = user("Jacky","993116",31,"男");
	user woman = user("风萧萧","aop794",22,"女");
	man.LeaveMessage("大家好!");
	woman.LeaveMessage("你好");
```



```
	return 0;
}
```

程序运行结果如图 11.15 所示。

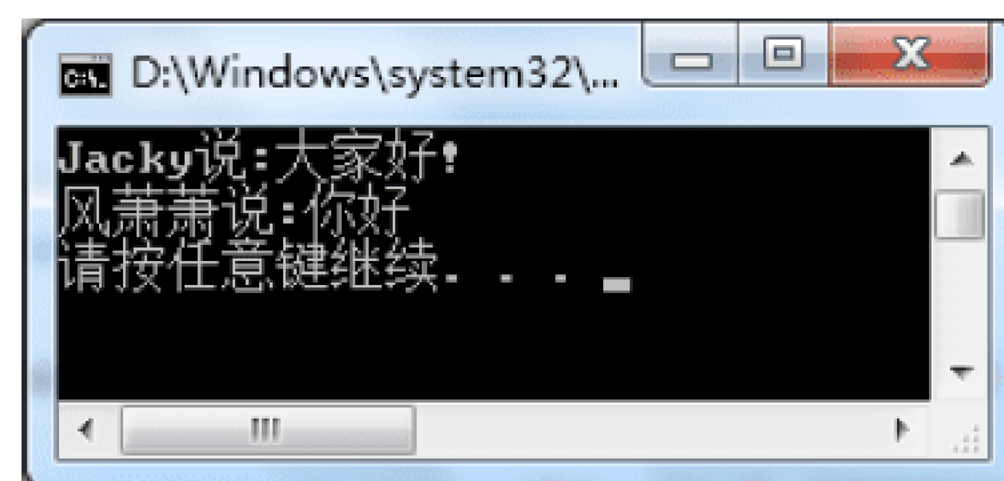


图 11.15 用户与留言

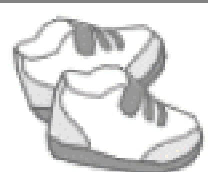


11.12.2 挑选硬盘

【例 11.13】 硬盘中比较重要的性能指标是容量和转速。现在首先挑选容量性价比较高的硬盘，其次挑选转速较高的。本实例设计一个硬盘类，在类中使用比较运算符>重载作为挑选硬盘的标准。在重载函数中比较容量价格和速度。代码如下：

实例位置：光盘\MR\Instance\11\11.13

```
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;
class hardDisk
{
public:
	int m_speed;			//速度
	int m_cap;			//容量
	int m_cost;			//价格
	hardDisk(int speed,int cap,int cost)
	{
		m_speed = speed;
		m_cap = cap;
		m_cost = cost;
	}
	bool operator>(const hardDisk& disk)
	{
		//性价比判断条件
		if((m_cap/m_cost)>(disk.m_cap/disk.m_cost))
		{
			return true;
		}
		else if((m_cap/m_cost)==(disk.m_cap/disk.m_cost) && m_speed>disk.m_speed)
		{
			return true;
		}
		return false;
	}
};
int main(int argc, _TCHAR* argv[])
{
	//作为示例过程，实质上买东西的情况会更复杂
	hardDisk* pDisk = NULL;
```





Note

```
    hardDisk diskArray[5] =
    {
        hardDisk(250,400,300),hardDisk(250,500,400),
        hardDisk(500,500,400),hardDisk(500,600,500),
        hardDisk(250,200,500)
    };
    pDisk = &diskArray[0];
    for(int i = 1;i<5;i++)
    {
        if(diskArray[i]>*pDisk)
        {
            pDisk = &diskArray[i];
        }
    }
    cout<<"挑了一款价格为"
        <<pDisk->m_cost
        <<",容量为"
        <<pDisk->m_cap
        <<"转速为"
        <<pDisk->m_speed
        <<"的硬盘"<<endl;
    return 0;
}
```

程序运行结果如图 11.16 所示。



图 11.16　挑选硬盘

11.13　本章常见错误

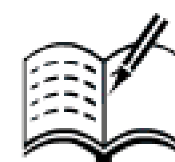
11.13.1　声明类时提示编译错误

在声明一个类或结构体时，如果没有在类结束的花括号后加上分号，那么在编译时就会产生错误。例如下面声明类少“;”就会出现错误。

```
class Test
{
public:
    void display();
}    //括号后面没有“;”，产生错误！
```



11.13.2　对比 const 与#define



const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换时可能会产生意料不到的错误。

有些集成化的调试工具可以对 const 常量进行调试，但是不能对宏常量进行调试。

11.13.3　new 和 delete 要配对使用

在程序中使用 new 运算符在堆中开辟一个空间之后，该内存使用完成后，一定要使用 delete 释放在堆中开辟的空间，否则会出现内存泄露。

11.14　本 章 小 结

通过本章的学习，使读者进入了有关面向对象的程序设计。在面向对象的程序设计中，其设计思路和人们日常生活中处理问题的方法相同，类是实现面向对象程序设计的基础。在本章中介绍了有关 C++中类的基础概念、如何声明类、如何实现一个类、构造函数和析构函数的作用以及类成员的相关内容，最后讲解了运算符重载应用。

11.15　跟 我 上 机

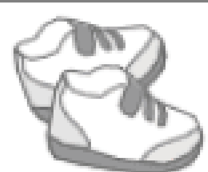
参考答案：光盘\MR\跟我上机

设计学生类，实现创建、修改和显示学生信息等功能。学生类包括学生姓名、性别、学号等。创建学生信息，学号 1、姓名 AA、性别男，修改为学号 5、姓名 CC、性别女。实现如下：

//student.h

```
#include <string>
class Student
{
	int num;                                        //学号
	std::string name;                               //姓名
	char sex;                                       //年龄
public:
	Student(int num,std::string name,char sex);     //添加学生信息
	void set(int num);                              //修改学号
	void set(std::string);                          //重载 set 函数，修改姓名
	void set(char sex);                             //修改性别
	void dis();                                     //显示

};
```





Note

//student.cpp

```
#include "student.h"
#include <iostream>
using namespace std;
Student::Student(int num,std::string name,char sex)
{
        this->name = name;
        this->sex = sex;
        this->num = num;
}
void Student::set(std::string name)
{
        this->name = name;
}
void Student::set(int num)
{
        this->num = num;
}
void Student::set(char sex)
{
        this->sex = sex;
}
void Student::dis()
{
        cout<<"学号："<<num<<endl
                <<"姓名："<<name<<endl;
        if(Student::sex == 'M')
                cout<<"性别："<<"男"<<endl;
        else if(Student::sex == 'W')
                cout<<"性别："<<"女"<<endl;
}
```

main.cpp

```
#include "student.h"
#include <iostream>
using namespace std;
int main()
{
        Student s(1,"AA",'M');              //用 Student 类实例化一个对象 s，创建学生信息
        s.dis();                            //显示学生信息
        cout<<"将学号修改为 5，姓名改为 CC，性别女\n";
        s.set(5);
        s.set("CC");
        s.set('W');                         //设置性别，M 为男，W 为女
        s.dis();
        return 0;
}
```

第12章

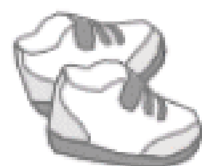
从基类到派生类

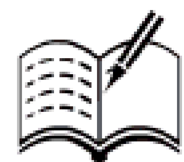
（视频讲解：46 分钟）

继承与派生是面向对象程序设计的两个重要特性，继承是从已有的类那里得到已有的特性，已有的类为基类或父类，新类被称为派生类或子类。继承与派生是从不同角度说明类之间的关系，这种关系包含了访问机制、多态和重载等。

本章能够完成的主要范例（已掌握的在方框中打勾）

- □ 以 3 种派生方式设计派生类
- □ 了解构造函数访问顺序
- □ 子类显示调用父类的构造函数
- □ 子类隐藏父类的成员函数
- □ 利用虚函数实现动态绑定
- □ 创建纯虚函数
- □ 设计学生类
- □ 设计类实现输出等边三角形和菱形的周长和面积





Note

12.1　类的继承

面向对象设计有 3 个主要特征：封装、继承和多态。本节重点介绍继承性（inheritance）。继承使得一个类可以从现有类中派生，而不必重新定义一个新类。继承的实质就是用已有的数据类型创建新的数据类型，并保留已有数据类型的特点，在既有类基础上创建新类，新类包含了旧类的数据成员和成员函数，并且可以在新类中添加新的数据成员和成员函数。旧类被称为基类或父类，新类被称为派生类或子类。

12.1.1　定义派生类

类继承的形式如下：

```
class 派生类名标识符:[继承方式] 基类名标识符
{
        [访问控制修饰符:]
        [成员声明列表]
};
```

继承方式有 3 种类型，分别为公有型（public）、私有型（private）和保护型（protected），访问控制修饰符也是 public、private、protected 3 种类型，成员声明列表中包含类的成员变量及成员函数，是派生类新增的成员。:是一个运算符，表示基类和派生类之间的继承关系，如图 12.1 所示。

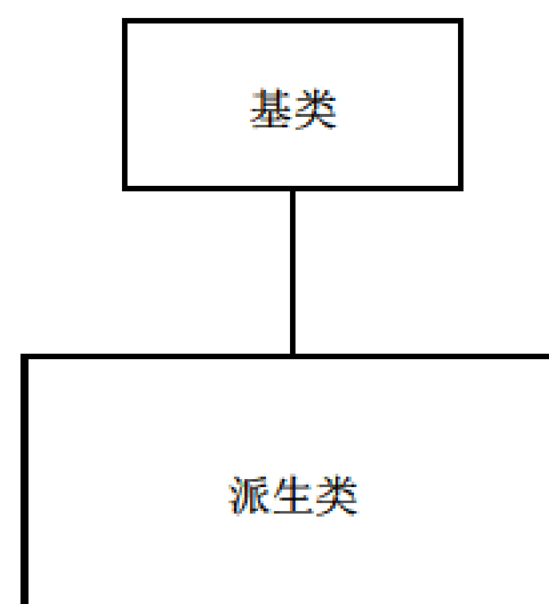


图 12.1　继承关系

例如，定义一个操作员类，使其继承于员工类。

定义一个员工类，它包含员工 ID、员工姓名、所属部门等信息。代码如下：

```
class CEmployee                              //定义员工类
{
public:
        int m_ID;                            //定义员工 ID
        char m_Name[128];                    //定义员工姓名
        char m_Depart[128];                  //定义所属部门
};
```

定义一个操作员类，通常操作员属于公司的员工，它包含员工 ID、员工姓名、所属部门等信息，此外还包含密码信息、登录方法等。代码如下：

```
class COperator :public CEmployee            //定义一个操作员类，从 CEmployee 类派生而来
{
public:
```



```
        char m_Password[128];                //定义密码
        bool Login();
};
```



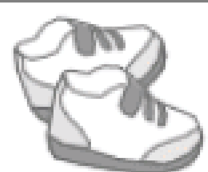
Note

操作员类是从员工类派生的一个新类，新类中增加密码信息、登录方法等信息，员工 ID、员工姓名等信息直接从员工类中继承得到。

【例 12.1】 以公有方式继承。

实例位置：光盘\MR\Instance\12\12.1

```
#include <iostream>
using namespace std;
class CEmployee                              //定义员工类
{
public:
        int m_ID;                            //定义员工 ID
        char m_Name[128];                    //定义员工姓名
        char m_Depart[128];                  //定义所属部门
        CEmployee()                          //定义默认构造函数
        {
                memset(m_Name,0,128);        //初始化 m_Name
                memset(m_Depart,0,128);      //初始化 m_Depart
}
void OutputName()                            //定义公有成员函数
{
        cout <<"员工姓名"<<m_Name<<endl;     //输出员工姓名
        }
};
class COperator :public CEmployee            //定义一个操作员类，以公有方式继承于 CEmployee
{
        public:
                char m_Password[128];        //定义密码
                bool Login()                 //定义登录成员函数
                {
                if (strcmp(m_Name,"MR")==0 &&    //比较用户名
                        strcmp(m_Password,"KJ")==0)  //比较密码
                {
                        cout<<"登录成功!"<<endl;     //输出信息
                        return true;                 //设置返回值
                }
                else
                {
                        cout<<"登录失败!"<<endl;     //输出信息
                        return false;                //设置返回值
                }
        }
};
int main(int argc, char* argv[])
{
        COperator optr;                      //定义一个 COperator 类对象
```



```
    strcpy(optr.m_Name,"MR");                //访问基类的 m_Name 成员
    strcpy(optr.m_Password,"KJ");            //访问 m_Password 成员
    optr.Login();                            //调用 COperator 类的 Login 成员函数
    optr.OutputName();                       //调用基类 CEmployee 的 OutputName 成员函数
    return 0;
}
```



Note

程序中 CEmployee 类是 COperator 类的基类，也就是父类。COperator 类将继承 CEmployee 类的所有非私有成员（private 类型成员不能被继承）。optr 对象初始化 m_Name 和 m_Password 成员后，调用了 Login 成员函数，程序运行结果如图 12.2 所示。



图 12.2　访问父类成员函数

用户在父类中派生子类时，可能存在一种情况，即在子类中定义一个与父类同名的成员函数，此时称为子类隐藏了父类的成员函数。例如，重新定义 COperator 类，添加一个 OutputName 成员函数。

12.1.2　访问类成员

类的特点之一就是具有封装性，封装在类里面的数据可以设置成对外可见或不可见，通过关键字 public、private、protected 可以设置类中数据成员对外是否可见，也就是其他类是否可以访问该数据成员。

关键字 public、private、protected 说明类成员是公有的、私有的还是保护的。这 3 个关键字将类划分为 3 个区域，在 public 区域的类成员可以在类作用域外被访问，而 private 区域和 protected 区域只能在类作用域内被访问，如图 12.3 所示。

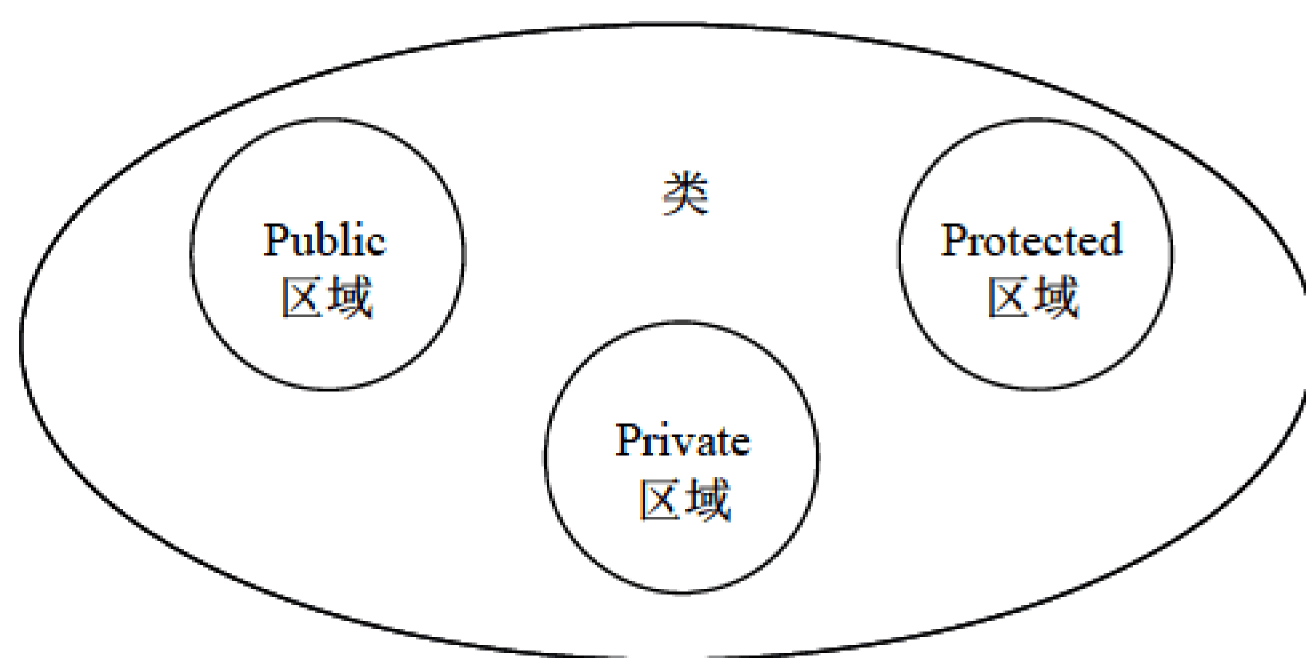


图 12.3　类成员属性

这 3 种类成员的属性如下：

☑　public 属性的成员对外可见，对内可见。

☑　private 属性的成员对外不可见，对内可见。

☑　protected 属性的成员对外不可见，对内可见，且对派生类是可见的。

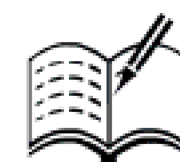
如果在类定义时没有加任何关键字，默认状态类成员都在 private 区域。

例如，在头文件 person.h 中：

```
class CPerson
{
    int m_iIndex;
    int getIndex()
{
return m_iIndex;
}
```



```
        int setIndex(int iIndex)
        {
                m_iIndex=iIndex;
                return 0;                                    //执行成功返回 0
        }
};
```



实现文件 person.cpp 中：

```
#include <iostream.h>
#include "Person.h"
void main()
{
        CPerson p;
        p.m_iIndex=100;                                      //错误，无法在类外访问私有成员
        cout << "m_iIndex is:" << p.getIndex() << endl;      //错误，同上
}
```

在编译上面的代码时，会发现编译不能通过，这是什么原因呢？

因为在默认状态下，类成员的属性为 private，这样的话类成员只能被类中的其他成员访问，而不能被外部访问。例如，CPerson 类中的 m_iIndex 数据成员，只能在类体的作用域内被访问和赋值，数据类型为 CPerson 类的对象 p，就无法对 m_iIndex 数据成员进行赋值。

有了不同区域，开发人员可以根据需求来进行封装。将不想让其他类访问和调用的类成员定义在 private 区域和 protected 区域，这就保证了类成员的隐蔽性。需要注意的是，如果将成员的属性设置为 protected，那么继承类也可以访问父类的保护成员，但不能访问类中的私有成员。

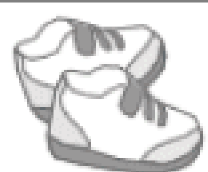
关键字的作用范围是，直到下一次出现另一个关键字为止，例如：

```
class CPerson
{
private:
        int m_iIndex;                                        //私有属性成员
public:
        int getIndex() { return m_iIndex; }                  //公有属性成员
        int setIndex(int iIndex)                             //公有属性成员
        {
                m_iIndex=iIndex;
                return 0;                                    //执行成功返回 0
        }
};
```

在上面的代码中，private 访问权限控制符设置 m_iIndex 成员变量为私有。public 关键字下面的成员函数设置为公有，从中可以看出 private 的作用域到 public 出现时为止。

12.1.3　类的派生方式

派生方式有 public、private、protected 3 种，下面分别进行介绍。





Note

☑　公有型派生

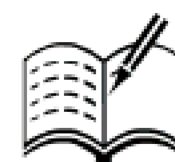
公有型派生表示对于基类中的 public 数据成员和成员函数，在派生类中仍然是 public，对于基类中的 private 数据成员和成员函数，在派生类中仍然是 private。例如：

```
class CEmployee
{
	public:
	void Output()
	{
		cout <<	m_ID << endl;
		cout <<	m_Name << endl;
		cout <<	m_Depart << endl;
	}
private:
	int m_ID;
	char m_Name[128];
	char m_Depart[128];
};
class COperator :public CEmployee
{
	void Output()
	{
		cout <<	m_ID << endl;			//引用基类的私有成员，错误
		cout <<	m_Name << endl;			//引用基类的私有成员，错误
		cout <<	m_Depart << endl;		//引用基类的私有成员，错误
		cout <<	m_Password << endl;		//正确
	}
private:
	char m_Password[128];
	bool Login();
};
```

COperator 类无法访问 CEmployee 类中的 private 数据成员 m_ID、m_Name 和 m_Depart，如果将 CEmployee 类中的所有成员都设置为 public 后，COperator 类才能访问 CEmployee 类中的所有成员。例如：

```
class CEmployee
{
public:
	void Output()
	{
		cout <<	m_ID << endl;
		cout <<	m_Name << endl;
		cout <<	m_Depart << endl;
	}
private:
	int m_ID;
	char m_Name[128];
	char m_Depart[128];
```





```
};
class COperator :public CEmployee
{
        void Output()
        {
                cout <<    m_ID << endl;                //正确
                cout <<    m_Name << endl;              //正确
                cout <<    m_Depart << endl;            //正确
                cout <<    m_Password << endl;          //正确
        }
private:
                char m_Password[128];
                bool Login();
};
```

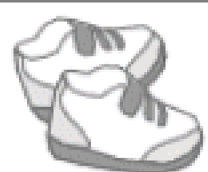
☑　私有型派生

私有型派生表示对于基类中的 public、protected 数据成员和成员函数，在派生类中可以访问。基类中的 private 数据成员，在派生类中不可以访问，例如：

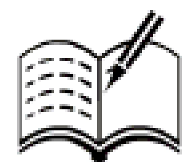
```
class CEmployee
{
public:
        void Output()
        {
                cout <<    m_ID << endl;
                cout <<    m_Name << endl;
                cout <<    m_Depart << endl;
        }
        int m_ID;
protected:
        char m_Name[128];
private:
        char m_Depart[128];
};
class COperator :private CEmployee
{
        void Output()
        {
                cout <<    m_ID << endl;                //正确
                cout <<    m_Name << endl;              //正确
                cout <<    m_Depart << endl;            //错误
                cout <<    m_Password << endl;          //正确
        }
private:
                char m_Password[128];
                bool Login();
};
```

☑　保护型派生

保护型派生表示对于基类中的 public、protected 数据成员和成员函数，在派生类中均为



protected。protected 类型在派生类的定义时可以访问，用派生类声明的对象不可以访问，也就是说在类体外不可以访问。protected 成员可以被基类的所有派生类使用。这一性质可以沿继承树无限向下传播。



Note

因为保护类的内部数据不能被随意更改，实例类本身负责维护，这就起到很好的封装性作用。把一个类分作两部分，一部分是公共的，另一部分是保护，保护成员对于使用者来说是不可见的，也是不需要了解的，这就减少了类与其他代码的关联程度。类的功能是独立的，它不依赖于应用程序的运行环境，既可以放到这个程序中使用，也可以放到那个程序中使用。这就能够非常容易地用一个类替换另一个类。类访问限制的保护机制使人们编制的应用程序更加可靠和易维护。

12.1.4　父类和子类的构造顺序

由于父类和子类中都有构造函数和析构函数，那么子类对象在创建时是父类先进行构造，还是子类先进行构造？同样在子类对象释放时，是父类先进行释放，还是子类先进行释放？这都有先后顺序。答案是当从父类派生一个子类并声明一个子类的对象时，它将先调用父类的构造函数，然后调用当前类的构造函数来创建对象；在释放子类对象时，先调用的是当前类的析构函数，然后是父类的析构函数。

【例 12.2】 构造函数访问顺序。

实例位置：光盘\MR\Instance\12\12.2

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class CEmployee                                          //定义 CEmployee 类
{
public:
	int m_ID;                                            //定义数据成员
	char m_Name[128];                                    //定义数据成员
	char m_Depart[128];                                  //定义数据成员
	CEmployee()                                          //定义构造函数
	{
		cout << "CEmployee 类构造函数被调用"<< endl;      //输出信息
	}
	~CEmployee()                                         //析构函数
	{
		cout << "CEmployee 类析构函数被调用"<< endl;      //输出信息
	}
};
class COperator :public CEmployee                        //从 CEmployee 类派生一个子类
{
public:
	char m_Password[128];                                //定义数据成员
	COperator()                                          //定义构造函数
	{
		strcpy(m_Name,"MR");                             //设置数据成员
```





Note

```
        cout << "COperator 类构造函数被调用"<< endl;        //输出信息
    }
    ~COperator()                                          //析构函数
    {
        cout << "COperator 类析构函数被调用"<< endl;        //输出信息
    }
};
int main(int argc, char* argv[])                          //主函数
{
    COperator optr;                                       //定义一个 COperator 对象
    return 0;
}
```

程序运行结果如图 12.4 所示。

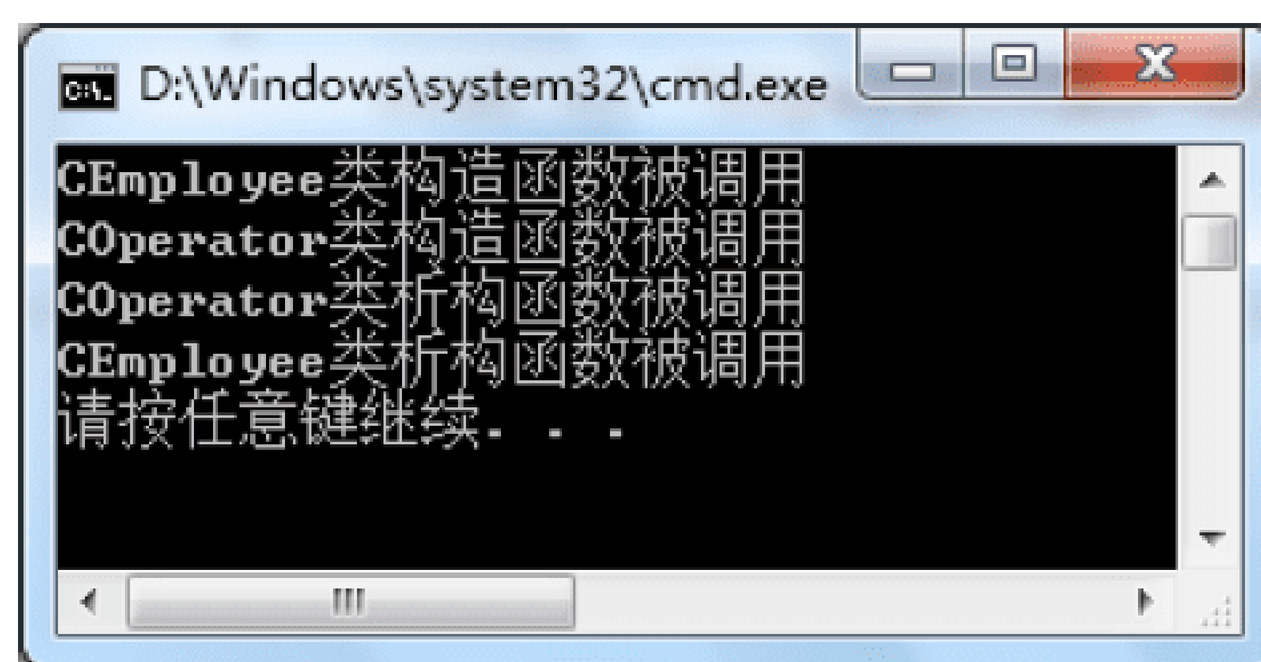


图 12.4　构造函数调用顺序

从图 12.4 中可以发现，在定义 COperator 类对象时，首先调用的是父类 CEmployee 的构造函数，然后是 COperator 类的构造函数。子类对象的释放过程则与其构造过程恰恰相反，先调用自身的析构函数，然后再调用父类的析构函数。

在分析完对象的构建、释放过程后，会考虑这样一种情况：定义一个基类类型的指针，调用子类的构造函数为其构建对象，当对象释放时，需要调用父类的析构函数还是先调用子类的析构函数，再调用父类的析构函数呢？答案是如果析构函数是虚函数，则先调用子类的析构函数，然后再调用父类的析构函数，如果析构函数不是虚函数，则只调用父类的析构函数。可以想象，如果在子类中为某个数据成员在堆中分配了空间，父类中的析构函数不是虚成员函数，将使子类的析构函数不被调用，其结果是对象不能被正确地释放，导致内存泄漏的产生。因此，在编写类的析构函数时，析构函数通常是虚函数。构造函数调用顺序不受基类在成员初始化表中是否存在以及被列出的顺序的影响。

12.1.5　子类显示调用父类构造函数

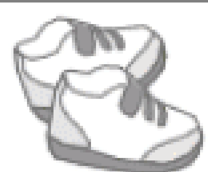
当父类含有带参数的构造函数时，子类创建时会调用它吗？答案是通过显式方式才可以调用。

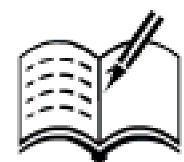
无论创建子类对象时调用的是哪种子类构造函数，都会自动调用父类默认构造函数。若想使用父类带参数的构造函数，则需要显式的方式。

【例 12.3】 子类显式调用父类的构造函数。

实例位置：光盘\MR\Instance\12\12.3

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class CEmployee                                           //定义 CEmployee 类
```






```
{
public:
        int m_ID;                                               //定义数据成员
        char m_Name[128];                                       //定义数据成员
        char m_Depart[128];                                     //定义数据成员
        CEmployee(char name[])                                  //带参数的构造函数
        {
            strcpy(m_Name,name);
            cout << m_Name<<"调用了 CEmployee 类带参数的构造函数"<< endl;
        }
        CEmployee()                                             //无参构造函数
        {
            strcpy(m_Name,"MR");
            cout << m_Name<<"CEmployee 类无参构造函数被调用"<< endl;
        }
        ~CEmployee()                                            //析构函数
        {
            cout << "CEmployee 类析构函数被调用"<< endl;          //输出信息
        }
};
class COperator:public CEmployee                                //从 CEmployee 类派生一个子类
{
public:
        char m_Password[128];//定义数据成员
        COperator(char name[ ]):CEmployee(name)                 //显示调用父类带参数的构造函数
        {           //设置数据成员
            cout << "COperator 类构造函数被调用"<< endl;           //输出信息
        }
        COperator():CEmployee("JACK")                           //显示调用父类带参数的构造函数
        {           //设置数据成员
            cout << "COperator 类构造函数被调用"<< endl;           //输出信息
        }
        ~COperator()                                            //析构函数
        {
            cout << "COperator 类析构函数被调用"<< endl;           //输出信息
        }
};
int main(int argc, char* argv[])                                //主函数
{
        COperator optr1;                    //定义一个 COperator 对象，调用自身无参构造函数
        COperator optr2("LaoZhang");        //定义一个 COperator 对象，调用自身带参数构造函数
        return 0;
}
```

程序运行结果如图 12.5 所示。

在父类无参构造函数中初始化成员字符串数组 m_Name 为 MR。从执行结果中看，子类对象创建时没有调用父类无参构造函数，调用的是带参数的构造函数。





Note

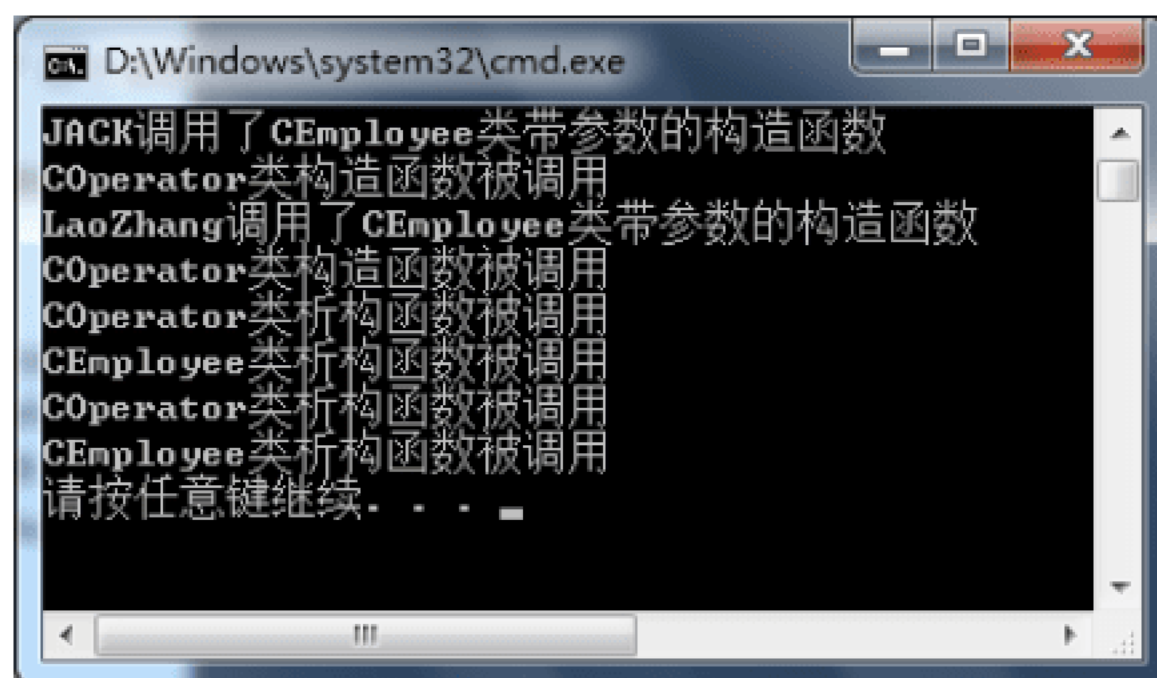


图 12.5　显式调用父类带参数构造函数



注意：

当父类只有带参数的构造函数时，子类必须以显式方法调用父类带参数的构造函数，否则编译会出现错误。

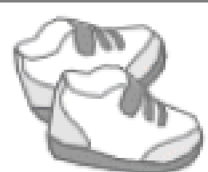
12.1.6　子类隐藏父类的成员函数

如果子类中定义了一个和父类一样的成员函数，那么是调用父类中的成员函数，还是调用子类中的成员函数呢？答案是调用子类中的成员函数。

【例 12.4】 子类隐藏父类的成员函数。

实例位置：光盘\MR\Instance\12\12.4

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class CEmployee                                         //定义 CEmployee 类
{
public:
        int m_ID;                                       //定义数据成员
        char m_Name[128];                               //定义数据成员
        char m_Depart[128];                             //定义数据成员
        CEmployee()                                     //定义构造函数
        {
                cout << "CEmployee 类构造函数被调用"<< endl;     //输出信息
        }
        ~CEmployee()                                    //析构函数
        {
                cout << "CEmployee 类析构函数被调用"<< endl;     //输出信息
        }
        void OutputName()
        {
                cout<< "调用 CEmployee 类的 OutputName  成员函数："<<endl;
        }
};
class COperator :public CEmployee                       //定义 COperator 类
{
```





Note

```
public:
    char m_Password[128];                          //定义数据成员
    void OutputName()                              //定义 OutputName 成员函数
    {
        cout << "操作员姓名: "<< m_Name<< endl;    //输出操作员姓名
    }
    bool Login()                                   //添加成员函数
    {
        if (strcmp(m_Name,"MR")==0 &&              //比较用户名
            strcmp(m_Password,"KJ")==0)            //比较密码
        {
            cout << "登录成功"<< endl;             //输出信息
            return true;                           //返回结果
        }
        else
        {
            cout << "登录失败"<< endl;             //输出信息
            return false;                          //返回结果
        }
    }
};
int main(int argc, char* argv[])                   //主成员函数
{
    COperator optr;                                //定义 COperator 对象
    strcpy(optr.m_Name,"MR");                      //设置 m_Name 数据成员
    optr.OutputName();                             //调用 COperator 类的 OutputName 成员函数
    return 0;
}
```

程序运行结果如图 12.6 所示。

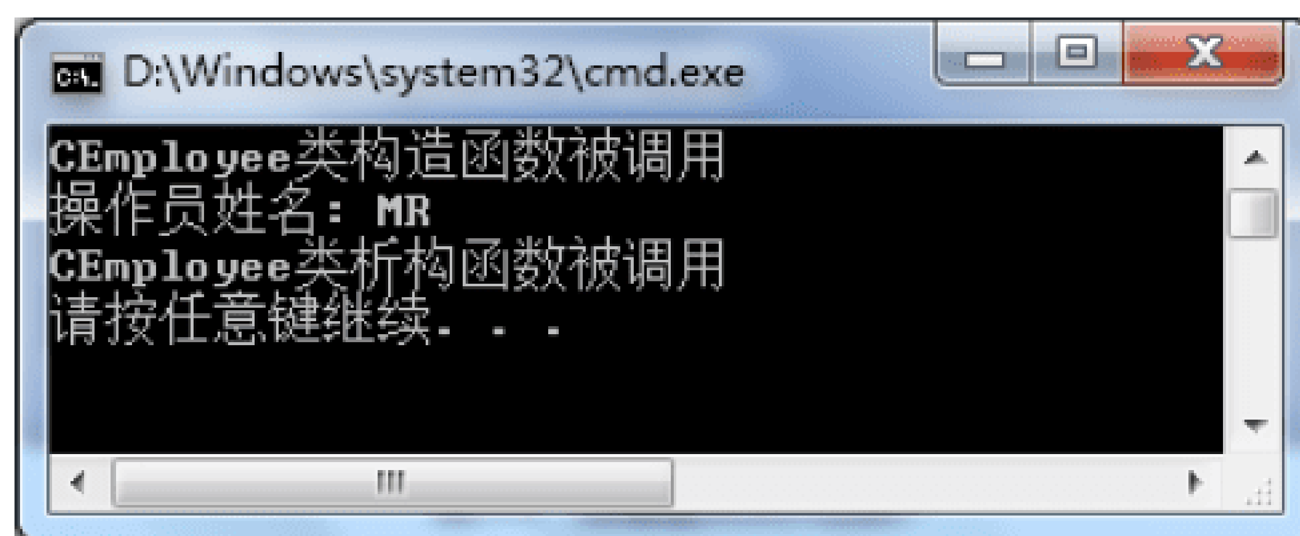
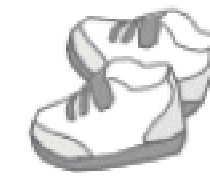


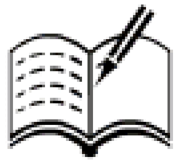
图 12.6　隐藏基类成员函数

从图 12.6 中可以发现，语句“optr.OutputName();”调用的是 COperator 类的 OutputName 成员函数，而不是 CEmployee 类的 OutputName 成员函数。如果用户想要访问父类的 OutputName 成员函数，需要显式使用父类名。例如：

```
COperator optr;                                    //定义一个 COperator 类
strcpy(optr.m_Name,"MR");                          //赋值字符串
optr.OutputName();                                 //调用 COperator 类的 OutputName 成员函数
optr.CEmployee::OutputName();                      //调用 CEmployee 类的 OutputName 成员函数
```

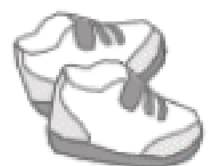
如果子类中隐藏了父类的成员函数，则父类中所有同名的成员函数（重载的函数）均被隐藏，





因此下面黑体部分代码是错误的。例如：

```
class CEmployee                                         //定义 CEmployee 类
{
public:
    int m_ID;                                           //定义数据成员
    char m_Name[128];                                   //定义数据成员
    char m_Depart[128];                                 //定义数据成员
    CEmployee()
    {
        memset(m_Name,0, 128);                          //初始化数据成员
        memset(m_Depart,0, 128);                        //初始化数据成员
        cout << "员工类构造函数被调用"<< endl;              //输出信息
    }
    void OutputName()                                   //定义重载成员函数
    {
        cout << "员工姓名: "<<m_Name<< endl;               //输出信息
    }
    void OutputName(const char* pchData)                //定义重载成员函数
    {
        if (pchData != NULL)                            //判断参数是否为空
        {
            strcpy(m_Name,pchData);                     //复制字符串
            cout << "设置并输出员工姓名:"<<pchData<< endl;  //输出信息
        }
    }
};
class COperator:public CEmployee                        //定义 COperator 类
{
public:
    char m_Password[128];                               //定义数据成员
    void OutputName()                   //定义 OutputName 成员函数，隐藏基类的成员函数
    {
        cout << "操作员姓名: "<<m_Name<< endl;             //输出信息
    }
    bool Login()                                        //定义 Login 成员函数
    {
        if (strcmp(m_Name,"MR")==0 &&                   //比较用户名称
        strcmp(m_Password,"KJ")==0)                     //比较用户密码
        {
            cout << "登录成功"<< endl;                    //输出信息
            return true;                                //设置返回值
        }
        else
        {
            cout << "登录失败"<< endl;                    //输出信息
            return false;                               //设置返回值
        }
```



```
    }
};
int main(int argc, char* argv[])
{
    COperator optr;                         //定义 COperator 类对象
    optr.OutputName("MR");                  //错误的代码，不能访问基类的重载成员函数
    return 0;
}
```



Note

程序中，在 CEmployee 类中定义了重载的 OutputName 成员函数，而在 COperator 类中又定义了一个 OutputName 成员函数，导致父类中的所有同名成员函数被隐藏。语句“optr.OutputName("MR");”是错误的。如果用户想要访问被隐藏的父类成员函数，依然需要指定父类名称。例如：

```
COperator optr;                             //定义一个 COperator 对象
optr.CEmployee::OutputName("MR");           //调用基类中被隐藏的成员函数
```

在派生完一个子类后，可以定义一个父类的类型指针，通过子类的构造函数为其创建对象。例如：

```
CEmployee *pWorker = new COperator ();      //定义 CEmployee 类型指针，调用子类构造函数
```

如果使用 pWorker 对象调用 OutputName 成员函数，例如，执行“pWorker->OutputName();”语句，调用的是 CEmployee 类的 OutputName 成员函数还是 COperator 类的 OutputName 成员函数呢？答案是调用 CEmployee 类的 OutputName 成员函数。编译器对 OutputName 成员函数进行的是静态绑定，即根据对象定义时的类型来确定调用哪个类的成员函数。由于 pWorker 属于 CEmployee 类型，因此调用的是 CEmployee 类的 OutputName 成员函数。那么是否有成员函数执行“pWorker->OutputName();”语句调用 COperator 类的 OutputName 成员函数呢？答案是通过定义虚函数可以实现。虚函数将会在后面章节讲到。

12.1.7 嵌套定义多个类

C++语言允许在一个类中定义另一个类，这被称为嵌套类。例如，下面的代码在定义 CList 类时，在内部又定义了一个嵌套类 CNode：

```
#define MAXLEN 128                          //定义一个宏
class CList                                 //定义 CList 类
{
public:                                     //嵌套类为公有的
    class CNode                             //定义嵌套类 CNode
    {
        friend class CList;                 //将 CList 类作为自己的友元类
    private:
        int m_Tag;                          //定义私有成员
```






```
        public:
            char m_Name[MAXLEN];                    //定义公有数据成员
        };                                          //CNode 类定义结束
public:
        CNode m_Node;                               //定义一个 CNode 类型数据成员
        void SetNodeName(const char *pchData)       //定义成员函数
        {
            if (pchData != NULL)                    //判断指针是否为空
            {
                strcpy(m_Node.m_Name,pchData);      //访问 CNode 类的公有数据
            }
        }
        void SetNodeTag(int tag)                    //定义成员函数
        {
            m_Node.m_Tag = tag;                     //访问 CNode 类的私有数据
        }
};
```

上述的代码在嵌套类 CNode 中定义了一个私有成员 m_Tag，定义了一个公有成员 m_Name，对于外围类 CList 来说，通常它不能够访问嵌套类的私有成员，虽然嵌套类是在其内部定义的。但是，上述代码在定义 CNode 类时将 CList 类作为自己的友元类，这使得 CList 类能够访问 CNode 类的私有成员。

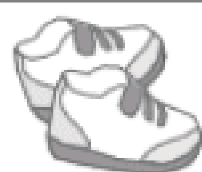
对于内部的嵌套类来说，只允许其在外围的类域中使用，在其他类域或者作用域中是不可见的。例如下面的定义是非法的：

```
int main(int argc, char* argv[])
{
        CNode node;                                 //错误的定义，不能访问 CNode 类
        return 0;
}
```

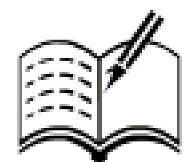
上述代码在 main 函数的作用域中定义了一个 CNode 对象，导致 CNode 没有被声明的错误。对于 main 函数来说，嵌套类 CNode 是不可见的，但是可以通过使用外围的类域作为限定符来定义 CNode 对象。如下的定义将是合法的：

```
int main(int argc, char* argv[])
{
        CList::CNode node;                          //合法的定义
        return 0;
}
```

上述代码通过使用外围类域作为限定符访问到了 CNode 类，但这样做通常是不合理的，也是有限制条件的。因为既然定义了嵌套类，通常都不允许在外界访问，这违背了使用嵌套类的原则。其次，在定义嵌套类时，如果将其定义为私有的或受保护的，即使使用外围类域作为限定符，外界也无法访问嵌套类。



12.2 多 重 继 承



Note

前文介绍的继承方式属于单继承，即子类只从一个父类继承公有的和受保护的成员。与其他面向对象语句不同，C++语言允许子类从多个父类继承公有的和受保护的成员，这被称为多重继承。多继承可以看作是单继承的扩展。一个派生类具有多个基类，派生类与每个基类之间的关系仍可看作是一个单继承。一个子类可以同时继承于多个父类，一个父类也可以被多个子类继承。

12.2.1 声明多重继承的派生类

多重继承是指有多个基类名标识符，其声明形式如下：

```
class 派生类名标识符: [继承方式] 基类名标识符 1,⋯,访问控制修饰符 基类名标识符 n
{
    [访问控制修饰符:]
    [成员声明列表]
};
```

声明形式中有:运算符，基类名标识符之间用“,”运算符分开。

例如，鸟能够在天空飞翔，鱼能够在水里游，而水鸟既能够在天空飞翔，又能够在水里游。那么在定义水鸟类时，可以将鸟和鱼同时作为其基类。

【例 12.5】 派生类的多重继承。

实例位置：光盘\MR\Instance\12\12.5

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class CBird                                         //定义鸟类
{
public:
    void FlyInSky()                                 //定义成员函数
    {
        cout << "鸟能够在天空飞翔"<< endl;          //输出信息
    }
    void Breath()                                   //定义成员函数
    {
        cout << "鸟能够呼吸"<< endl;                //输出信息
    }
};
class CFish                                         //定义鱼类
{
public:
    void SwimInWater()                              //定义成员函数
    {
```






```
            cout << "鱼能够在水里游"<< endl;          //输出信息
        }
        void Breath()                                  //定义成员函数
        {
            cout << "鱼能够呼吸"<< endl;              //输出信息
        }
};
class CWaterBird: public CBird, public CFish           //定义水鸟，从鸟和鱼类派生
{
public:
        void Action()                                  //定义成员函数
        {
            cout << "水鸟既能飞又能游"<< endl;        //输出信息
        }
};
int main(int argc, char* argv[])                       //主函数
{
        CWaterBird waterbird;                          //定义水鸟对象
        waterbird.FlyInSky();                          //调用从鸟类继承而来的 FlyInSky 成员函数
        waterbird.SwimInWater();                       //调用从鱼类继承而来的 SwimInWater 成员函数
        return 0;
}
```

程序运行结果如图 12.7 所示。



图 12.7　多重继承

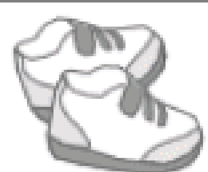
程序中定义了鸟类 CBird，定义了鱼类 Cfish，然后从鸟类和鱼类派生了一个子类水鸟类 CWaterBird。水鸟类自然继承了鸟类和鱼类的所有公有和受保护的成员，因此 CWaterBird 类对象能够调用 FlyInSky 和 SwimIn Water 成员函数。在 CBird 类中提供了一个 Breath 成员函数，在 CFish 类中同样提供了 Breath 成员函数，如果 CWaterBird 类对象调用 Breath 成员函数，将会执行哪个类的 Breath 成员函数呢？答案是将会出现编译错误，编译器将产生歧义，不知道具体调用哪个类的 Breath 成员函数。为了让 CWaterBird 类对象能够访问 Breath 成员函数，需要在 Breath 成员函数前具体指定类名。例如：

```
waterbird.CFish::Breath();                             //调用 CFish 类的 Breath 成员函数
waterbird.CBird::Breath();                             //调用 CBird 类的 Breath 成员函数
```

在多重继承中存在这样一种情况，假如 CBird 类和 CFish 类均派生于同一个父类，例如 CAnimal 类，那么当从 CBird 类和 CFish 类派生子类 CWaterBird 时，在 CWaterBird 类中将存在两个 CAnimal 类的复制。能否在派生 CWaterBird 类时，使其只存在一个 CAnimal 基类呢？为了解决该问题，C++语言提供了虚继承的机制，虚继承将会在后面章节讲到。

12.2.2　注意避免二义性

派生类在调用成员函数时，先在自身的作用域内寻找，如果找不到，会到基类中寻找，但当



派生类继承的基类中有同名成员时，派生类中就会出现来自不同基类的同名成员。例如：



Note

```
class CBaseA
{
public:
	void function();
};
class CBaseB
{
public:
	void function();
};
class CDeriveC:public CBaseA,public CBaseB
{
public:
	void function();
};
```

CBaseA 和 CBaseB 都是 CDeriveC 的父类，并且两个父类中都含有 function 成员函数，CDeriveC 将不知道调用哪个基类的 function 成员函数，这就产生了二义性。当使用多继承时就会容易产生二义性。

12.2.3　多重继承的构造顺序

前面讲过，单一继承是先调用基类的构造函数，然后调用派生类的构造函数，但多重继承将如何调用构造函数呢？多重继承中的基类构造函数被调用的顺序以类派生表中声明的顺序为准。派生表就是多重继承定义中继承方式后面的内容，调用顺序就是按照基类名标识符的前后顺序进行的。

【例 12.6】 多重继承的构造顺序。

实例位置：光盘\MR\Instance\12\12.6

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class CBicycle                                  //声明一个 CBicycle 类
{
public:
	CBicycle()                                  //构造函数
	{
		cout << "Bicycle Construct" << endl;
	}
	CBicycle(int iWeight)                       //重载构造函数
	{
		m_iWeight=iWeight;
	}
	void Run()
```





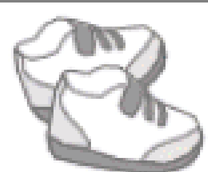
```
        {
            cout << "Bicycle Run" << endl;
        }
protected:
        int m_iWeight;                              //保护类型成员变量
};
class CAirplane                                     //声明一个 CAirplane 类
{
public:
        CAirplane()                                 //构造函数
        {
            cout << "Airplane Construct" << endl;
        };
        CAirplane(int iWeight)
        {
            m_iWeight=iWeight;
        }
        void Fly()
        {
            cout << "Airplane Fly" << endl;
        }
protected:
        int m_iWeight;
};
//声明一个派生类，以公有方式继承于 CBicycle 和 CAirplane
class CAirBicycle : public CBicycle, public CAirplane
{
public:
        CAirBicycle()                               //构造函数
        {
            cout << "CAirBicycle Construct" << endl;
        }
        void RunFly()
        {
            cout << "Run and Fly" << endl;
        }
};
void main()
{
        CAirBicycle ab;                             //用 CAirBicycle 类定义一个对象
        ab.RunFly();                                //通过对象调用 RunFly 函数
}
```

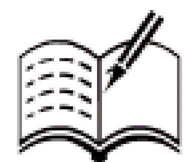
程序运行结果如图 12.8 所示。

程序中基类的声明顺序是先 CBicycle 类后 CAirplane 类，所以对象的构造顺序就是先 CBicycle 类后 CAirplane 类，最后是 CAirBicycle 类。



图 12.8　多重继承的构造顺序





Note

12.3　C++的多态性

多态性（polymorphism）是面向对象程序设计的一个重要特征，利用多态性可以设计和实现一个易于扩展的系统。在C++语言中，多态性是指具有不同功能的函数可以用同一个函数名，这样就可以用一个函数名调用不同内容的函数，发出同样的消息被不同类型的对象接收时，导致完全不同的行为。这里所说的消息主要指类的成员函数的调用，而不同的行为是指不同的实现。

多态性通过联编实现。联编是指一个计算机程序自身彼此关联的过程。按照联编所进行的阶段不同，可分为两种不同的联编方法：静态联编和动态联编。在C++中，根据联编的时刻不同，存在两种类型多态性，即函数重载和虚函数。

12.3.1　虚函数概述

C++规定动态联编是在虚函数的支持下实现的，虚函数是动态联编的基础。虚函数是非静态成员函数。在类的继承层次结构中，在不同的层次中可以出现名字、参数个数和类型都相同而功能不同的函数。编译器按照先自己后父类的顺序进行查找覆盖，如果子类有父类相同原型的成员函数时，要想调用父类的成员函数，需要对父类重新引用调用。虚函数则可以解决子类和父类相同原型成员函数的函数调用问题。虚函数允许在派生类中重新定义与基类同名的函数，并且可以通过基类指针或引用来访问基类和派生类中的同名函数。

在基类中用 virtual 声明成员函数为虚函数，在派生类中重新定义此函数，改变该函数的功能。在C++语言中虚函数可以继承，当一个成员函数被声明为虚函数后，其派生类中的同名函数都自动成为虚函数，但如果派生类没有覆盖基类的虚函数，则调用时调用基类的函数定义。

覆盖和重载的区别是：重载是同一层次函数名相同，覆盖是在继承层次中成员函数的函数原型完全相同。

12.3.2　动态绑定

多态主要体现在虚函数上，只要有虚函数存在，对象类型就会在程序运行时动态绑定。动态绑定的实现方法是定义一个指向基类对象的指针变量，并使它指向同一类族中需要调用该函数的对象，通过该指针变量调用此虚函数。动态绑定中，子类有的函数调用子类的，子类没有的调用父类的。

【例 12.7】 利用虚函数实现动态绑定。

实例位置：光盘\MR\Instance\12\12.7

```
#include "stdafx.h"
#include <iostream>
using namespace std;
```





```
class CEmployee                                  //定义 CEmployee 类，作为父类
{
public:
        int m_ID;                                //定义数据成员
        char m_Name[128];                        //定义数据成员
        char m_Depart[128];                      //定义数据成员
        CEmployee()                              //定义构造函数
        {
                memset(m_Name,0,128);            //初始化数据成员
                memset(m_Depart,0,128);          //初始化数据成员
        }
        virtual void OutputName()                //定义一个虚成员函数
        {
                cout << "员工姓名: "<<m_Name << endl;     //输出信息
        }
};
class COperator :public CEmployee                //从 CEmployee 类派生一个子类
{
public:
        char m_Password[128];                    //定义数据成员
        void OutputName()                        //重载父类的 OutputName 函数
        {
                cout << "操作员姓名: "<<m_Name<< endl;    //输出信息
        }
};
int main(int argc, char* argv[])
{
        //定义 CEmployee 类型指针，调用 COperator 类构造函数构造一个 COperator 的对象
        CEmployee *pWorker = new COperator();
        strcpy(pWorker->m_Name,"MR");            //设置 m_Name 数据成员信息
        pWorker->OutputName();                   //调用 COperator 类的 OutputName 成员函数
        delete pWorker;                          //释放对象
        return 0;
}
```

上述代码中，在 CEmployee 类中定义了一个虚函数 OutputName，在子类 COperator 中改写了 OutputName 成员函数，其中 COperator 类中的 OutputName 成员函数即使没有使用 virtual 关键字仍为虚函数。定义一个 CEmployee 类型的指针，调用 COperator 类的构造函数构造对象。

程序运行结果如图 12.9 所示。



图 12.9　虚函数

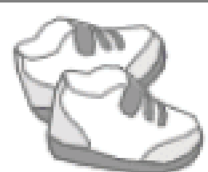
从图 12.9 中可以发现，“pWorker-> OutputName();”语句调用的是 COperator 类的 OutputName 成员函数。虚函数有以下几方面的限制：

☑　只有类的成员函数才能为虚函数。

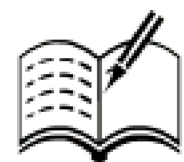
☑　静态成员函数不能是虚函数，因为静态成员函数不受限于某个对象。

☑　内联函数不能是虚函数，因为内联函数是不能在运行中动态确定其位置的。

☑　构造函数不能是虚函数，析构函数通常是虚函数。



12.3.3 虚继承机制



Note

12.2.1 节讲到从 CBird 类和 CFish 类派生子类 CWaterBird 时，在 CWaterBird 类中将存在两个 CAnimal 类的复制。那么如何在派生 CWaterBird 类时使其只存在一个 CAnimal 基类呢？C++语言提供的虚继承机制解决了这个问题。

【例 12.8】 虚继承。

实例位置：光盘\MR\Instance\12\12.8

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class CAnimal                                   //定义一个动物类
{
public:
        CAnimal()                               //定义构造函数
        {
            cout << "动物类被构造"<< endl;      //输出信息
        }
        void Move()                             //定义成员函数
        {
            cout << "动物能够移动"<< endl;      //输出信息
        }
};
class CBird : virtual public CAnimal            //从 CAnimal 类虚继承 CBird 类
{
public:
        CBird()                                 //定义构造函数
        {
            cout << "鸟类被构造"<< endl;        //输出信息
        }
        void FlyInSky()                         //定义成员函数
        {
            cout << "鸟能够在天空飞翔"<< endl;  //输出信息
        }
        void Breath()                           //定义成员函数
        {
            cout << "鸟能够呼吸"<< endl;        //输出信息
        }
};
class CFish: virtual public CAnimal             //从 CAnimal 类虚继承 CFish 类
{
public:
        CFish()                                 //定义构造函数
        {
            cout << "鱼类被构造"<< endl;        //输出信息
```






```
    }
    void SwimInWater()                          //定义成员函数
    {
        cout << "鱼能够在水里游"<< endl;           //输出信息
    }
    void Breath()                               //定义成员函数
    {
        cout << "鱼能够呼吸"<< endl;              //输出信息
    }
};
class CWaterBird: public CBird, public CFish    //从 CBird 和 CFish 类派生子类 CWaterBird
{
public:
    CWaterBird()                                //定义构造函数
    {
        cout << "水鸟类被构造"<< endl;            //输出信息
    }
    void Action()                               //定义成员函数
    {
        cout << "水鸟既能飞又能游"<< endl;        //输出信息
    }
};
int main(int argc, char* argv[])                //主函数
{
    CWaterBird waterbird;                       //定义水鸟对象
    return 0;
}
```

程序运行结果如图 12.10 所示。

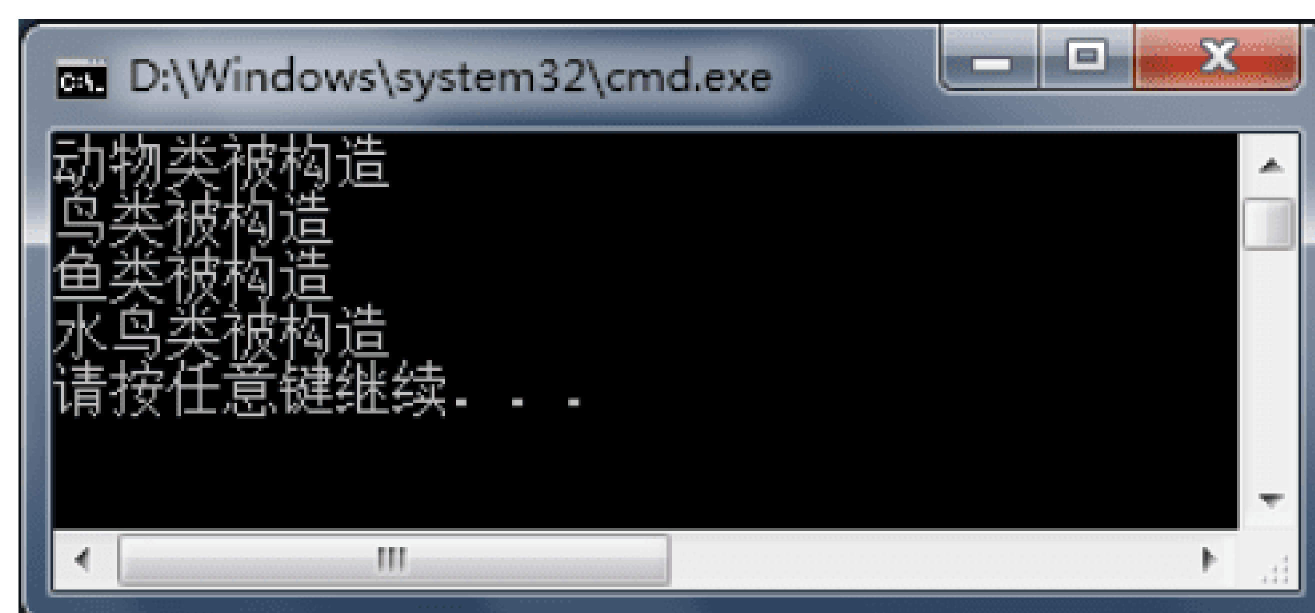
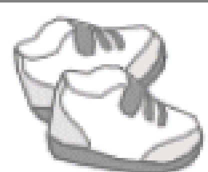


图 12.10　虚继承

上述代码在定义 CBird 类和 CFish 类时使用了关键字 virtual 从基类 CAnimal 派生而来。实际上，虚继承对于 CBird 类和 CFish 类没有多少影响，却对 CWaterBird 类产生了很大影响。CWaterBird 类中不再有两个 CAnimal 类的复制，而只存在一个 CAnimal 的复制。

通常，在定义一个对象时，先依次调用基类的构造函数，最后才调用自身的构造函数。但是对于虚继承来说，情况有些不同。在定义 CWaterBird 类对象时，先调用基类 CAnimal 的构造函数，然后调用 CBird 类的构造函数，这里 CBird 类虽然为 CAnimal 的子类，但是在调用 CBird 类的构造函数时将不再调用 CAnimal 类的构造函数。对于 CFish 类也是同样的道理。

在程序开发过程中，多继承虽然带来了很多方便，但是很少有人愿意使用它，因为多继承会带来很多复杂的问题，并且它能够完成的功能通过单继承同样可以实现。如今流行的 C#、Delphi、Java 等面向对象语言没有提供多继承的功能而是只采用单继承是经过设计者充分考虑的。因此，读者在开发应用程序时，如果能够使用单继承实现，尽量不要使用多继承。



12.4　抽象类介绍



Note

包含有纯虚函数的类称为抽象类，一个抽象类至少具有一个纯虚函数。抽象类只能作为基类派生出的新的子类，而不能在程序中被实例化（即不能说明抽象类的对象），但是可以使用指向抽象类的指针。在开发程序过程中并不是所有代码都是由软件构造师自己写的，有时需要调用库函数，有时需要分给别人写。一名软件构造师可以通过纯虚函数建立接口，然后让程序员填写代码实现接口，而自己主要负责建立抽象类。

12.4.1　创建纯虚函数

纯虚函数（pure virtual function）是指被标明为不具体实现的虚成员函数，它不具备函数的功能。许多情况下，在基类中不能给虚函数一个有意义的定义，这时可以在基类中将它说明为纯虚函数，而其实现留给派生类去做。纯虚函数不能被直接调用，仅起到提供一个与派生类相一致的接口的作用。声明纯虚函数的形式为：

```
virtual   类型  函数名(参数表列)= 0;
```

纯虚函数不可以被继承。当基类是抽象类时，在派生类中必须给出基类中纯虚函数的定义，或在该类中再声明其为纯虚函数。只有在派生类中给出了基类中所有纯虚函数的实现时，该派生类才不再成为抽象类。抽象类是不能定义对象的，在实际中为了强调一个类是抽象类，可将该类的构造函数声明为保护的控制权限。

【例 12.9】 创建纯虚函数。

实例位置：光盘\MR\Instance\12\12.9

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class CFigure
{
public:
	virtual double getArea() = 0;                    //声明一个纯虚函数
};
const double PI=3.14;
class CCircle : public CFigure                       //派生类 CCircle
{
private:
	double m_dRadius;
public:
	CCircle(double dR){m_dRadius=dR;}
	double getArea()                                 //给出纯虚函数的定义
	{
```





```
            return m_dRadius*m_dRadius*PI;
        }
};
class CRectangle : public CFigure                //派生类 CRectangle
{
protected:
        double m_dHeight,m_dWidth;
public:
        CRectangle(double dHeight,double dWidth)
        {
            m_dHeight=dHeight;
            m_dWidth=dWidth;
        }
        double getArea()                          //给出纯虚函数的定义
        {
            return m_dHeight*m_dWidth;
        }
};
void main()
{
        CFigure *fg1;
        fg1= new CRectangle(4.0,5.0);
        cout << fg1->getArea() << endl;
        delete fg1;
        CFigure *fg2;
        fg2= new CCircle(4.0);
        cout << fg2->getArea() << endl;
        delete fg2;
}
```

程序运行结果如图 12.11 所示。

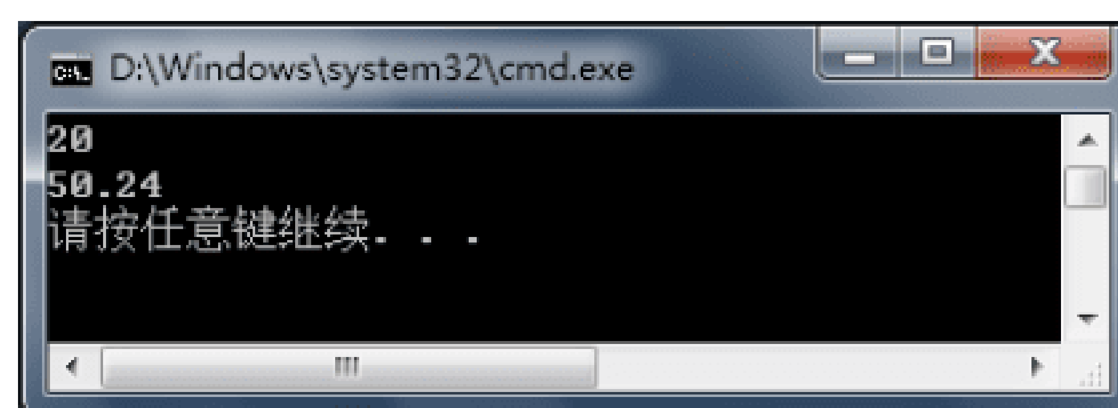


图 12.11　创建纯虚函数

程序定义了矩形类 CRectangle 和圆形类 CCircle，两个类都派生于图形类 CFigure。图形类是一个在现实生活中不存在的对象，抽象类面积的计算方法不确定，所以，将图形类 CFigure 的面积计算方法设置为纯虚函数，这样圆形有圆形面积的计算方法，矩形有矩形面积的计算方法，每个继承自 CFigure 的对象都有自己的面积，通过 getArea 成员函数就可以获取面积值。

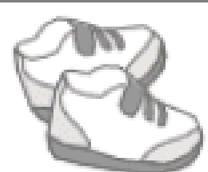


注意：

对于包含纯虚函数的类来说，是不能够实例化的，“CFigure figure;”是错误的。

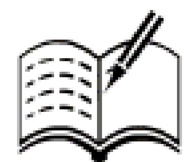
12.4.2　实现抽象类中的成员函数

抽象类通常用于作为其他类的父类，从抽象类派生的子类如果不是抽象类，则子类必须实现



父类中的所有纯虚函数。

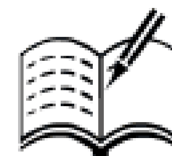
【例 12.10】 实现抽象类中的纯虚函数。



实例位置：光盘\MR\Instance\12\12.10

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class CEmployee                                    //定义 CEmployee 类
{
public:
	int m_ID;                                      //定义数据成员
	char m_Name[128];                              //定义数据成员
	char m_Depart[128];                            //定义数据成员
	virtual void OutputName() = 0;                 //定义抽象成员函数
};
class COperator :public CEmployee                  //定义 COperator 类，派生于 CEmployee 类
{
public:
	char m_Password[128];                          //定义数据成员
	void OutputName()                              //实现父类中的纯虚成员函数
	{
		cout << "操作员姓名: "<<m_Name<< endl;     //输出信息
	}
	COperator()                                    //定义 COperator 类的默认构造函数
	{
		strcpy(m_Name,"MR");                       //设置数据成员 m_Name 信息
	}
};
class CSystemManager :public CEmployee             //定义 CSystemManager 类
{
public:
	char m_Password[128];                          //定义数据成员
	void OutputName()                              //实现父类中的纯虚成员函数
	{
		cout << "系统管理员姓名: "<<m_Name<< endl; //输出信息
	}
	CSystemManager()                               //定义 CSystemManager 类的默认构造函数
	{
		strcpy(m_Name,"SK");                       //设置数据成员 m_Name 信息
	}
};
int main(int argc, char* argv[])                   //主函数
{
	CEmployee *pWorker;                            //定义 CEmployee 类型指针对象
	pWorker = new COperator();                     //调用 COperator 类的构造函数为 pWorker 赋值
	pWorker->OutputName();                         //调用 COperator 类的 OutputName 成员函数
	delete pWorker;                                //释放 pWorker 对象
	pWorker = NULL;                                //将 pWorker 对象设置为空
	//调用 CSystemManager 类的构造函数为 pWorker 赋值
```





Note

```
        pWorker = new CSystemManager();
        //调用 CSystemManager 类的 OutputName 成员函数
        pWorker->OutputName();
        delete pWorker;                          //释放 pWorker 对象
        pWorker = NULL;                          //将 pWorker 对象设置为空
        return 0;
}
```

程序中从 CEmployee 类派生了两个子类，分别为 COperator 和 CSystemManager。这两个类分别实现了父类的纯虚成员函数 OutputName。同样的一条语句“pWorker->OutputName();”，由于 pWorker 指向的对象不同，其行为也不同。程序运行结果如图 12.12 所示。



图 12.12　实现抽象类中的成员函数

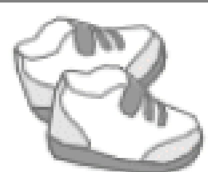
12.5　综 合 应 用

12.5.1　学生类的设计

【例 12.11】 现有一年级学生类包含姓名、年龄、学号、班级等属性。本实例实现利用类的派生设计一个二年级学生类，该类继承于一年级学生类。二年级类有一年级学生类的特性也有自己的特性。学生通过自我介绍，将这些属性展现出来。它们的共性有姓名、年龄、学号、班级，并且都可以作自我介绍。二年级拥有自己的专业属性，在自我介绍函数中可以先调用父类函数，再将专业信息加入。代码如下：

实例位置：光盘\MR\Instance\12\12.11

```
class grade1                                     //声明一年级类
{
protected:
        int m_class;                             //班级
        string m_name;                           //姓名
        int m_dAge;                              //年龄
        string m_dID;                            //学号
public:
        grade1(int c,string name,int age,string ID)
        {                                        //一年级类构造函数
                m_class = c;
                m_name = name;
```





Note

```
            m_dAge =age ;
            m_dID = ID;
        }
        virtual void introduce()                  //虚函数
        {
            cout<<"我是来自"<<m_class<<"班的"<<m_name<<","<<m_dAge
                <<"岁"<<","<<"学号是"<<m_dID<<endl;
        }
};
class grade2:public grade1                        //派生一个二年级类
{
private:
        string m_sp;                              //新增专业成员变量
public:
        grade2(int c,string name,int age,string ID,string sp):grade1(c,name,age,ID)
        {
            m_sp = sp;
        }
        virtual void introduce()                  //重载虚函数
        {
            cout<<"我是来自"<<m_class<<"班的"<<m_sp<<"专业的"<<m_name<<","
                <<m_dAge<<"岁"<<","<<"学号是"<<m_dID<<endl;
        }
};
int main(int argc, _TCHAR* argv[])
{
        grade1 g1 = grade1(3,"Lin",20,"yk1032838");
        g1.introduce();
        grade2 g2 = grade2(5,"Sam",21,"yk2813631","电子信息");
        g2.introduce();
        return 0;
}
```

程序运行结果如图 12.13 所示。

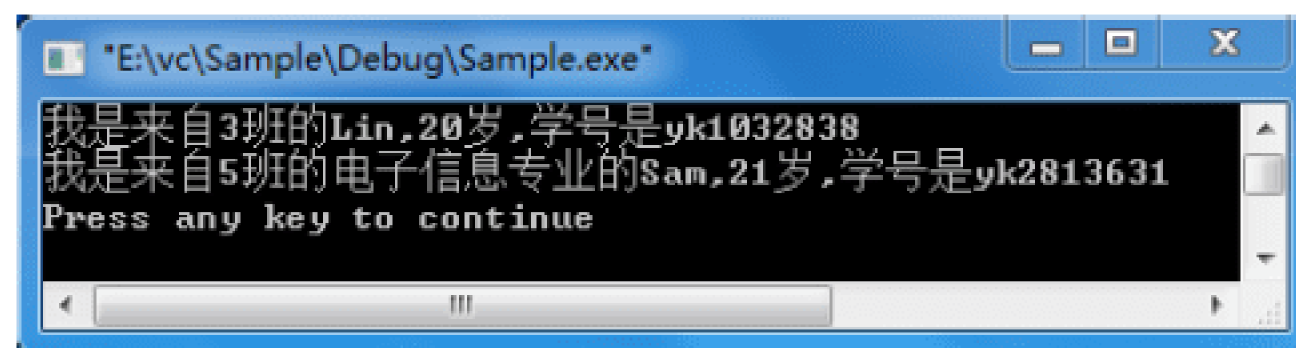


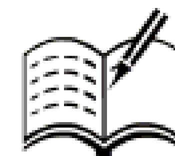
图 12.13　学生类的设计

12.5.2　等边多边形

【例 12.12】 等边三角形与正方形都是等边多边形。多边形具有边长、面积、周长等相同属性。本实例设计 3 个类：多边形、正方形和三角形，添加相应的属性，并实现方法能够输出等边三角形和菱形的周长和面积。

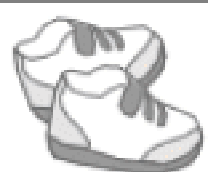
多边形都具有边长这个属性。通过边长可以求出面积和周长，但不同种类的多边形计算公式不相同。可以将等边多边形定义为一个抽象类。等边三角形和菱形继承多边形类，实现求面积和周长的函数。关键代码如下（等边三角形的面积计算公式可以取 0.433 倍的边长的平方）：





实例位置：光盘\MR\Instance\12\12.12

```
class polygon                                  //等边多边形类
{
protected:
    float m_dSide;                             //边长
public:
    virtual void getArea() = 0;                //计算周长
    virtual void getPerimeter() = 0;           //计算面积
};
class triangle :public polygon                 //等边三角形
{
public:
    triangle(int k)
    {
        m_dSide = k;
    }
    void getArea()                             //计算等边三角形面积
    {
        cout<<"这个等边三角形的面积为:"<<0.433*m_dSide*m_dSide<<endl;
    }
    void getPerimeter()                        //计算等边三角形周长
    {
        cout<<"这个等边三角形的周长为:"<< 3.0*m_dSide <<endl;
    }
};
class square :public polygon                   //正方形
{
public:
    square(int k)                              //构造函数给边长初始化
    {
        m_dSide = k;
    }
    void getArea()                             //计算正方形面积
    {
        cout<<"这个等边三角形的面积为:"<<m_dSide*m_dSide<<endl;
    }
    void getPerimeter()                        //计算正方形周长
    {
        cout<<"这个等边三角形的周长为:"<< 4.0*m_dSide <<endl;
    }
};
int main(int argc, _TCHAR* argv[])
{
    square s =square(3);
    s.getArea();
    s.getPerimeter();
    triangle t =triangle(4);
    t.getArea();
    t.getPerimeter();
```



```
	return 0;
}
```

程序运行结果如图 12.14 所示。



Note

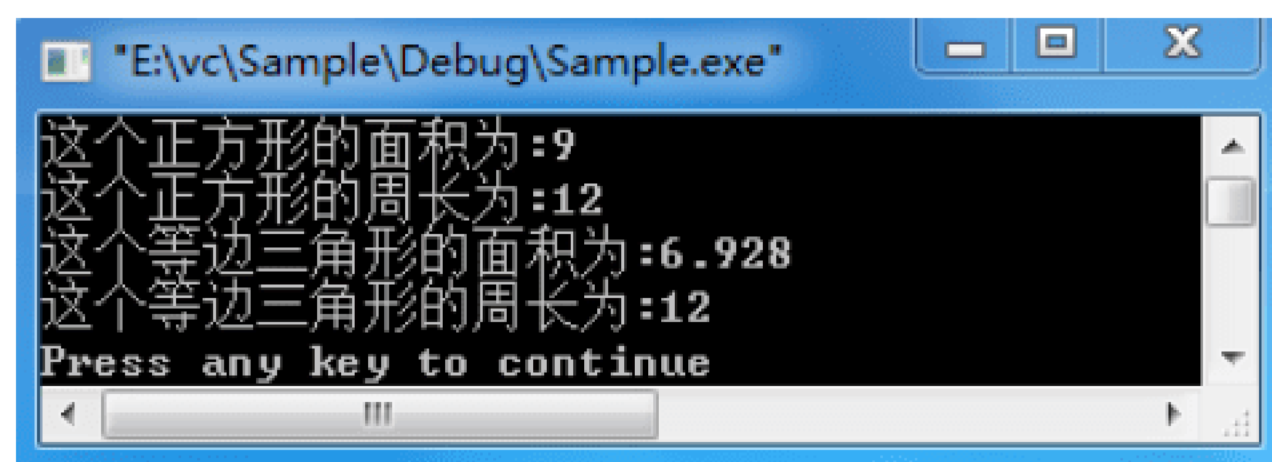


图 12.14　等边多边形

12.5.3　教师职位信息

【例 12.13】 设计一个程序，分别定义教师类（Teacher）和职位类（Level），采用多重继承方式由这两个类派生出新类 Teacher_Level（教师和职位信息）。要求如下：

（1）在 Teacher 类中包含“职称”数据成员，Level 类中包含“职务”。在 Teacher_Level 类中还包含“工资”数据成员。

（2）在 Teacher_Level 类中使用 Show 函数将信息输出。

代码如下：

实例位置：光盘\MR\Instance\12\12.13

```
#include "stdafx.h"
#include<iostream>
#include<string>
using namespace std;
class Teacher                                   //定义教师任课类
{
public:
	Teacher(string);                        //进行初始化
protected:
	string title;                           //职称
};
Teacher::Teacher( string t)
{
	title=t;
}
class Leader                                    //定义职务级别类
{
public:
	Leader( string p);                      //初始化
protected:
	string postion;                         //职位
};
Leader::Leader( string p)
{
	postion=p;
}
//多重继承
class Teacher_Leader: public Leader, public Teacher
{
```





```
public:
	Teacher_Leader(string t,string p,string w):Teacher(t),Leader(p),wage(w){};
	void Show();
protected:
	string wage;                                          //工资
};
void Teacher_Leader::Show()                               //定义成员函数
{
	cout<<"职务是："<<title<<endl;
	cout<<"职位是："<<postion<<endl;
	cout<<"工资是："<<wage<<endl;
}
int _tmain(int argc, _TCHAR* argv[])
{
	Teacher_Leader person("语文","教员","3500");           //初始化对象
	person.Show();
	return 0;
}
```

程序运行结果如图 12.15 所示。



图 12.15　教师职位信息

12.6　本章常见错误

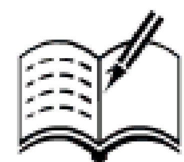
12.6.1　静态成员函数不能访问普通成员变量

普通成员函数在参数传递时编译器会附加一个隐含的 this 指针，通过 this 指针来确定调用哪个对象中的成员变量。但是静态成员函数没有 this 指针。所以在程序中不可以用静态成员函数访问类中的普通变量，它只能访问静态成员变量。

12.6.2　类初始化时不能直接给数组名赋值

```
class A
{
	int x;
	char str[10];
```

```
public:
    A(int n,char *p)
    {
        x = n;              //正确
        str = p;            //错误，数组名是常量，不可直接给数组名赋值
        strcpy(str,p);      //正确
    }
};
```


Note

12.6.3 派生后的访问权限总结

公有派生		私有派生		保护派生	
基类属性	派生类权限	基类属性	派生类权限	基类属性	派生类权限
私有	不能访问	私有	不能访问	私有	不能访问
保护	保护	保护	私有	保护	保护
公有	公有	公有	私有	公有	保护

12.7 本章小结

本章介绍了面向对象程序设计中的关键技术——继承与派生，继承和派生在使用上还涉及二义性、访问顺序等许多技术问题，正确理解和处理这些技术有利于掌握继承的使用方法。继承中还涉及多重继承，这增加了面向对象开发的灵活性。面向对象可以建立抽象类，由抽象类派生新类，可以形成对类的一定管理。最后介绍了友元类和友元函数的使用方法。

12.8 跟我上机

参考答案：光盘\MR\跟我上机

设计一个学生类 Student，包含学生姓名、学号、分数，实现显示学生信息和平均成绩功能。再设计一个研究生类 Graduate，以公有方式继承学生类的成员，并添加自己的新成员学位等级 graduateName，并重载学生类的 displayStuInfo 方法。

实例化 3 个 Graduate 对象，显示它们的信息并求平均分。实现如下：

student.h

```
#include <string>
using std::string;
class Student                       //学生类
{
```





```
public:
        static int total_num;
        static double total_score;
        int num;
        double score;
        std::string name;
        Student(string name = "no name",int n = 0,double s = 0);
        virtual void displayStuInfo();
        void displyAverage();
};
```

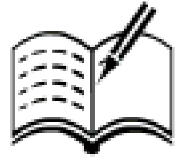
student.cpp

```
#include "stdafx.h"
#include "Student.h"
#include <iostream>
using std::cout;
using std::endl;
using std::string;
int Student::total_num = 0;
double Student::total_score = 0;
Student::Student(string name,int n,double s):name(name),num(n),score(s)
{
        total_num++;
        total_score += score;
}
void Student::displyAverage()
{
//      double average = total_score / total_num;
        cout<<"平均分："<<total_score / total_num<<endl;
}
void Student::displayStuInfo()
{
        cout<<"姓名："<<name<<endl;
        cout<<"学号："<<num<<endl;
        cout<<"分数："<<score<<endl;
}
```

graduate.h

```
#include "Student.h"
class Graduate:public Student  //研究生类，继承于学生类
{
        string graduateName;
public:
        Graduate(string name = "no name",int n = 0,double s = 0,
               string graName = "NO Graduate");
        void displayStuInfo();
};
```





Note

graduate.cpp

```
#include "stdAfx.h"
#include "graduate.h"
#include <iostream>
using std::cout;
using std::endl;
Graduate::Graduate(string name,int n,double s,string graName):
Student(name,n,s),graduateName(graName)
{
}
void Graduate::displayStuInfo()
{
	cout<<"姓名："<<name<<endl;
	cout<<"学号："<<num<<endl;
	cout<<"分数："<<score<<endl;
	cout<<"Graduate:"<<graduateName<<endl;
}
```

main.cpp

```
#include "stdafx.h"
#include "Graduate.h"
#include <iostream>
using std::cout;
using std::endl;
int _tmain(int argc, _TCHAR* argv[])
{
	Graduate gra1("AAA",1,90,"硕士");
	Graduate gra2("BBB",2,80,"硕士");
	Graduate gra3("CCC",3,85,"硕士");
	gra1.displayStuInfo();
	gra2.displayStuInfo();
	gra3.displayStuInfo();
	cout<<"\n 三人平均分："<<endl;
	gra1.displyAverage();
//	gra2.displyAverage();
	return 0;
}
```

第 13 章

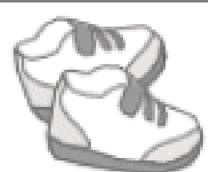
C++模板的使用

（视频讲解：50 分钟）

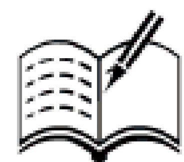
模板是 C++的高级特性，分为函数模板和类模板。对于程序员来说，要完全掌握 C++模板的用法并不容易。模板使程序员能够快速建立具有类型安全的类库集合和函数集合，它的实现大大方便了大规模软件开发。本章将介绍 C++模板的基本概念、函数模板和类模板，使读者有效地掌握模板的用法，正确使用 C++系统日益庞大的标准模板库。

本章能够完成的主要范例（已掌握的在方框中打勾）

- □ 使用数组作为模板参数
- □ 求字符串的最小值
- □ 为具体类型的参数提供默认值
- □ 使用 assert 进行越界警告
- □ 定制类模板成员函数
- □ 简单链表的实现
- □ 使用 CList 类模板
- □ 在类模板中使用静态数据成员



13.1 函数模板



函数模板不是一个实在的函数，编译器不能为其生成可执行代码。定义函数模板后只是一个对函数功能框架的描述，当它具体执行时，将根据传递的实际参数决定其功能。

13.1.1 定义函数模板

函数模板定义的一般形式如下：

```
template <类型形式参数表> 返回类型 函数名(形式参数表)
{
…    //函数体
}
```

template 为关键字，表示定义一个模板，尖括号<>表示模板参数，模板参数主要有两种，一种是模板类型参数，另一种是模板非类型参数。上述代码中定义的模板使用的是模板类型参数，模板类型参数使用关键字 class 或 typedef 开始，其后是一个用户定义的合法标识符。模板非类型参数与普通参数定义相同，通常为一个常数。

可以将声明函数模板分成 template 部分和函数名部分。例如：

```
template<class T>
void fun(T t)
{
…    //函数实现
}
```

定义一个求和的函数模板，例如：

```
template <class type>                    //定义一个模板类型
type Sum(type xvar,type yvar)            //定义函数模板
{
    return xvar + yvar;
}
```

在定义完函数模板之后，需要在程序中调用函数模板。下面的代码演示了 Sum 函数模板的调用：

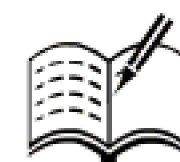
```
int iret = Sum(10,20);                   //实现两个整数的相加
double dret = Sum(10.5,20.5);            //实现两个实数的相加
```

如果采用如下的形式调用 Sum 函数模板，将会出现错误：



```
int iret = Sum(10.5,20);                //错误的调用，type 类型不统一
double dret = Sum(10,20.5);             //错误的调用
```

上述代码中为函数模板传递了两个类型不同的参数，编译器产生了歧义。如果用户在调用函数模板时显式标识模板类型，就不会出现错误了。例如：



```
int iret = Sum<int>(10.5,20);           //正确地调用函数模板
double dret = Sum<double>(10,20.5);     //正确地调用函数模板
```

用函数模板生成实际可执行的函数又称为模板函数。函数模板与模板函数不是一个概念。从本质上讲，函数模板是一个“框架”，它不是真正可以编译生成代码的程序，而模板函数是把函数模板中的类型参数实例化后生成的函数，它和普通函数本质是相同的，可以生成可执行代码。

13.1.2　使用函数模板

假设求两个函数之中最大者，如果想求整型数和实型数需要定义两个函数，两个函数定义如下：

```
int max(int a，int b)
{
    return a>b?a：b；                   //返回最小值
}
float max(float a，float b)
{
    return a>b?a：b；                   //返回最小值
}
```

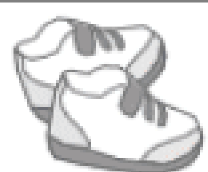
能不能通过一个 max 函数来完成既求整型数之间最大者又求实型数之间最大者呢？答案是使用函数模板以及#define 宏定义。

#define 宏定义可以在预编译期对代码进行替换。例如：

```
#define max(a,b) ((a) > (b) ? (a) : (b))    //条件选择
```

上述代码可以求整数最大值和实型数最大值。但宏定义#define 只是进行简单替换，它无法对类型进行检查，有时计算结果可能不是预计的，例如：

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
using namespace std;
#define max(a,b) ((a) > (b) ? (a) : (b))
void main()
{
     int m=0,n=0;
     cout << max(m,++n) << endl;         //使用宏
     cout << m << setw(2) << endl;       //输出 m，按两个宽度输出
}
```





Note

程序运行结果如图 13.1 所示。

程序运行的预期结果应该是 1 和 0，为什么输出这样的结果呢？原因在于宏替换之后“++n”被执行了两次，因此 n 的值是 2 不是 1。

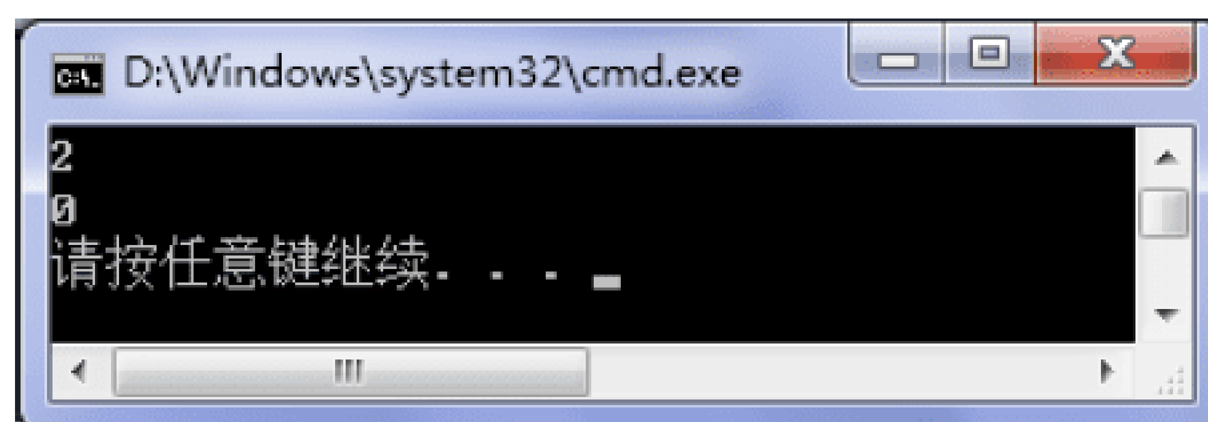


图 13.1　利用宏定义求最大值

宏是预编译指令，很难调试，无法单步进入宏的代码中。模板函数和#define 宏定义相似，但模板函数是用模板实例化得到的函数，它与普通函数没有本质区别，可以重载模板函数。

使用模板求最大值的代码如下：

```
template<class Type>                    //定义模板类型
Type max(Type a,Type b)                 //定义函数模板
{
    if(a > b)
        return a;                       //返回最大的数
    else
        return b;
}
```

调用模板函数 max 可以正确计算整型数和实型数最大值。例如：

```
cout << "最大值：" << max(10,1) << endl;
cout << "最大值：" << max(200.05,100.4) << endl;
```

【例 13.1】 使用数组作为模板参数。

实例位置：光盘\MR\Instance\13\13.1

```
#include <iostream>
using namespace std;
template <class type,int len>           //定义一个模板类型
type Max(type array[len])               //定义函数模板
{
    type ret = array[0];                //定义一个变量
    for(int i=1; i<len; i++)            //遍历数组元素
    {
        ret = (ret > array[i])? ret : array[i];    //比较数组元素大小
    }
    return ret;                         //返回最大值
}
void main()
{
    int array[5] = {1,2,3,4,5};         //定义一个整型数组
    int iret = Max<int,5>(array);       //调用函数模板 Max
    double dset[3] = {10.5,11.2,9.8};   //定义实数数组
    double dret = Max<double,3>(dset);  //调用函数模板 Max
    cout << dret << endl;
}
```

程序运行结果如图 13.2 所示。



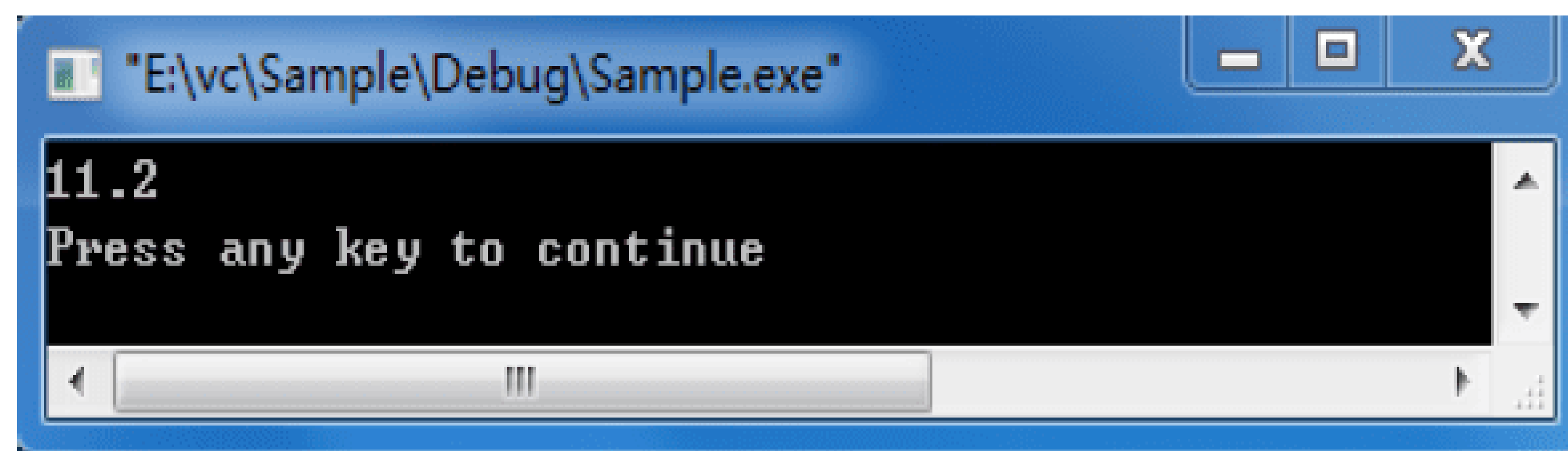
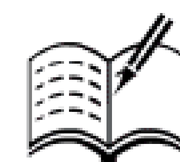


图 13.2　使用数组作为模板参数



程序中定义一个函数模板 Max，用来求数组中元素的最大值，其中模板参数使用模板类型参数 type 和模板非类型参数 len，参数 type 声明了数组中的元素类型，参数 len 声明了数组中的元素个数，给定数组元素后，程序将数组中的最大值输出。

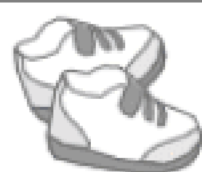
13.1.3　重载函数模板

整型数和实型数编译器可以直接进行比较，所以使用函数模板后也可以直接进行比较，但如果是字符指针指向的字符串该如何比较呢？答案是通过重载函数模板来实现。通常字符串需要库函数来进行比较，通过重载函数模板实现字符串的比较。

【例 13.2】 求出字符串的最小值。

实例位置：光盘\MR\Instance\13\13.2

```
#include "stdafx.h"
#include <iostream >
#include <string >
using namespace std;
template<class Type>
Type min(Type& a,Type& b)                        //定义函数模板
{
    if(a < b)
        return a;                                //如果 a 小返回 a，否则返回 b
    else
        return b;
}
char   min(char a,char b)                        //重载函数模板
{
    if(strcmp(&a,&b))
        return b;
    else
        return a;
}
void main ()
{
    cout << "最小值：" << min(10,1) << endl;
    cout << "最小值：" << min('a','b') << endl;
    string s1 = "hi";
    string s2 = "mr";
    cout << "最小值：" << min(s1,s2) << endl;
}
```





Note

程序运行结果如图 13.3 所示。



图 13.3　求出字符串的最小值

程序在重载的函数模板 min 的实现中，使用 strcmp 库函数来完成字符串的比较，此时使用 min 函数可以比较整型数据、实型数据、字符数据和字符串数据。

13.2　类　模　板

使用 template 关键字不但可以定义函数模板，也可以定义类模板。类模板代表一族类，是用来描述通用数据类型或处理方法的机制，它使类中的一些数据成员和成员函数的参数或返回值可以取任意数据类型。类模板可以说是用类生成类，减少了类的定义数量。

13.2.1　定义类模板

类模板的一般定义形式为：

```
template <类型形式参数表> class 类模板名
{
…    //类模板体
};
```

类模板成员函数定义形式为：

```
template <类型形式参数表>
返回类型 类模板名 <类型名表>:::成员函数名(形式参数列表)
{
…    //函数体
}
```

template 是关键字，类型形式参数表与函数模板定义相同。类模板的成员函数定义时的类模板名与类模板定义时要一致，类模板不是一个真实的类，需要重新生成类，生成类的形式如下：

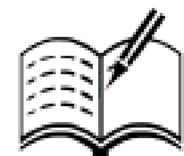
```
类模板名<类型形式参数表>
```

用新生成的类定义对象的形式如下：

```
类模板名<类型形式参数表> 对象名
```



其中类型在参数表应与该类模板中的类型形式参数表匹配。用类模板生成的类称为模板类。类模板和模板类不是同一个概念，类模板是模板的定义，不是真实的类，定义中要用到类型参数，模板类本质上与普通类相同，它是类模板的类型参数实例化之后得到的类。



定义一个容器的类模板，代码如下：

```
template<class Type>                        //定义模板类型
class Container                             //定义一个类
{
    Type tItem;                             //私有成员
public:
    Container(){};                          //构造函数
    void begin(const Type& tNew);           //函数模板
    void end(const Type& tNew);             //函数模板
    void insert(const Type& tNew);          //函数模板
    void empty(const Type& tNew);           //函数模板
};
```

和普通类一样，需要对类模板成员函数进行定义，代码如下：

```
void Container<type>:: begin (const Type& tNew)       //容器的第一个元素
{
    tItem=tNew;
}
void Container<type>:: end (const Type& tNew)         //容器的最后一个元素
{
    tItem=tNew;
}
void Container<type>::insert(const Type& tNew)        //向容器中插入元素
{
    tItem=tNew;
}
void Container<type>:: empty (const Type& tNew)       //清空容器
{
    tItem=tNew;
}
```

将模板类的参数设置为整型，然后用模板类声明对象。代码如下：

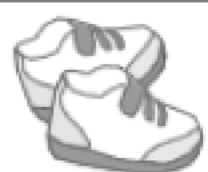
```
Container<int> myContainer;                 //声明 Container<int>类对象
```

声明对象后，就可以调用类成员函数，代码如下：

```
int i=10;
myContainer.insert(i);                      //调用 insert 函数
```

在类模板定义中，类型形式参数表中的参数也可以是其他类模板，例如：

```
template < template<class A> class B>
class CBase
```



```
{
private:
	B<int> m_n;
};
```



Note

类模板也可以进行继承，例如：

```
template <class T>
class CDerived public T
{
public:
	CDrived();
};
template <class T>
CDerived<T>::CDerived() : T()
{
	cout << "" <<endl;
}
void main()
{
	CDerived<CBase1> D1;
	CDerived<CBase1> D1;
}
```

T 是一个类，CDerived 继承自该类，CDerived 可以对类 T 进行扩展。

13.2.2　执行时指定参数

类模板中的类型形式参数表可以在执行时指定，也可以在定义类模板时指定。下面看类型参数如何在执行时指定。

【例 13.3】 简单类模板。

实例位置：光盘\MR\Instance\13\13.3

```
#include "stdafx.h"
#include <iostream>
using namespace std;
template<class T1,class T2>
class MyTemplate
{
	T1 t1;
	T2 t2;
public:
	MyTemplate(T1 tt1,T2 tt2)
	{t1 =tt1, t2=tt2;}
	void display()
	{ cout << t1 << ' ' << t2 << endl;}
};
```





```
void main()
{
	int a=123;
	double b=3.1415;
	MyTemplate<int ,double> mt(a,b);
	mt.display();
}
```

程序运行结果如图 13.4 所示。

程序中的 MyTemplate 是一个模板类，它使用整型类型和双精度作为参数。

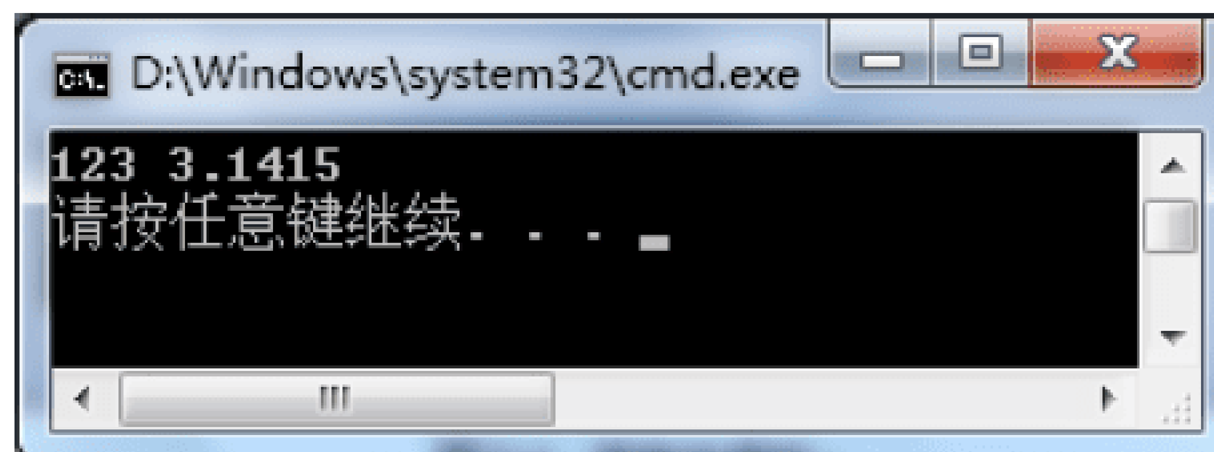


图 13.4　简单类模板

13.2.3　设置默认模板参数

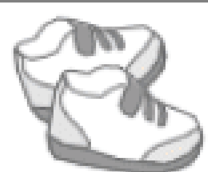
默认模板参数就是在类模板定义时设置类型形式参数表中一个类型参数的默认值，该默认值是一个数据类型，有默认的数据类型参数后，在定义模板新类时就可以不进行指定。

【例 13.4】 默认模板参数。

实例位置：光盘\MR\Instance\13\13.4

```
#include "stdafx.h"
#include <iostream>
using namespace std;
template <class T1,class T2 = int>                    //声明模板类型
class MyTemplate                                      //定义一个类 MyTemplate
{
	T1 t1;                                            //私有成员变量
	T2 t2;
public:
	MyTemplate(T1 tt1,T2 tt2)                         //构造函数
	{t1=tt1;t2=tt2;}
	void display()
	{
		cout<< t1 << ' ' << t2 << endl;
	}
};
void main()
{
	int a=123;
	double b=3.1415;
	MyTemplate<int ,double> mt1(a,b);                 //调用类模板，创建对象 mt1，并初始化
	MyTemplate<int> mt2(a,b);                         //创建对象 mt2
	mt1.display();                                    //通过对象 mt1 调用显示函数
	mt2.display();                                    //通过对象 mt2 调用显示函数
}
```

程序运行结果如图 13.5 所示。





Note

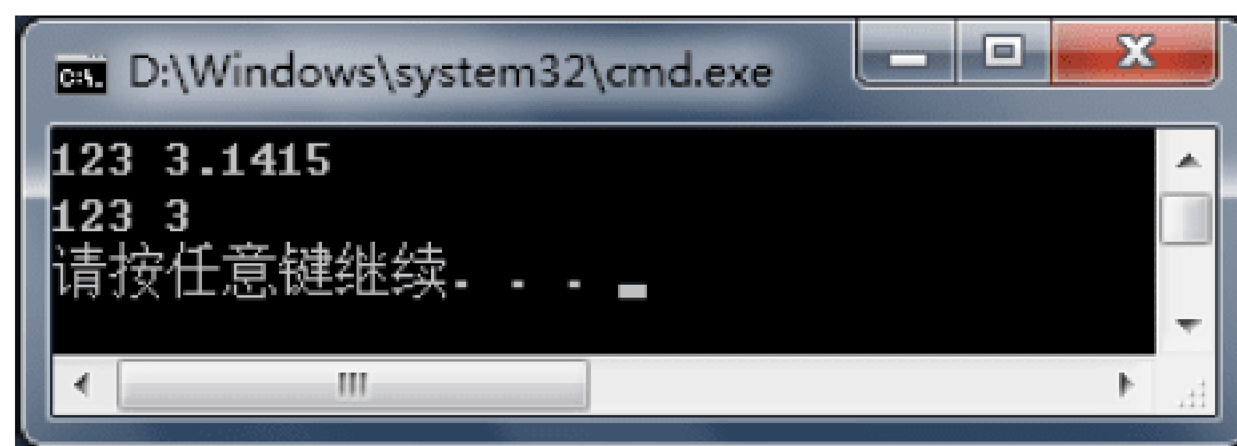


图 13.5　默认模板参数

13.2.4　为具体类型的参数提供默认值

默认模板参数是类模板中由默认的数据类型作参数，在模板定义时还可以为默认的数据类型声明变量，并且为变量赋值。

【例 13.5】 为具体类型的参数提供默认值。

实例位置：光盘\MR\Instance\13\13.5

```
#include "stdafx.h"
#include <iostream>
using namespace std;
template<class T1,class T2,int num= 10 >                    //定义模板类型
class MyTemplate                                            //定义类
{
	T1 t1;                                                  //私有对象成员
	T2 t2;
public:
	MyTemplate(T1 tt1,T2 tt2)                               //构造函数
	{t1 =tt1+num, t2=tt2+num;}
	void display()
	{ cout << t1 <<' ' << t2 <<endl;}
};
void main()
{
	int a=123;
	double b=3.1415;
	MyTemplate<int ,double> mt1(a,b);                       //用类模板实例化一个对象 mt1
	MyTemplate<int ,double ,100> mt2(a,b);                  //实例化对象 mt2
	mt1.display();                                          //输出 t1 和 t2
	mt2.display();
}
```

程序运行结果如图 13.6 所示。

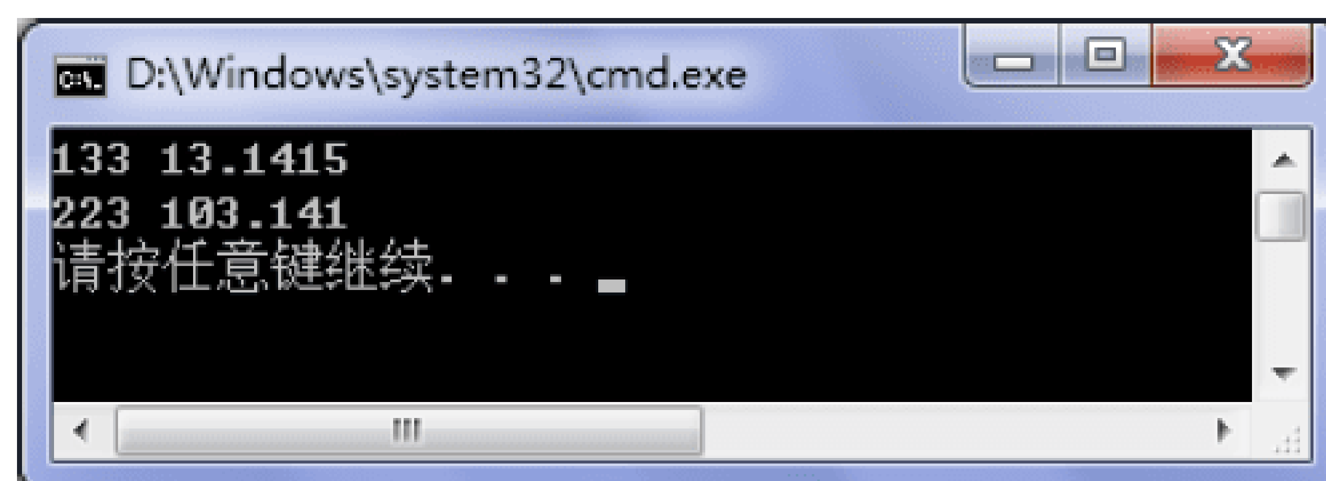


图 13.6　为具体类型的参数提供默认值





13.2.5　越界检测

C++语言不能检查数组下标是否越界，如果下标越界会造成程序崩溃，程序员在编辑代码时很难找到下标越界错误。那么如何能让数组进行下标越界检测呢？答案是建立数组模板，在模板定义时对数组的下标进行检查。

在模板中想要获取下标值，需要重载数组下标运算符[]，重载数组下标运算符后使用模板类实例化的数组，即可进行下标越界检测。例如：

```
#include <cassert>
template <class T,int b>
class Array
{
	T& operator[] (int sub)						//重载函数，引用作返回值
	{
		assert(sub>=0&& sub<b);					//调用 assert 来进行警告处理
	}
};
```

程序中使用了 assert 来进行警告处理，当有下标越界情况发生时就弹出对话框警告，然后输出出现错误的代码位置。assert 函数需要使用 cassert 头文件。

【例 13.6】 数组模板的应用示例：越界警告。

实例位置：光盘\MR\Instance\13\13.6

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <cassert>
using namespace std;
class Date
{
	int iMonth,iDay,iYear;
	char Format[128];
public:
	Date(int m=0,int d=0,int y=0)					//构造函数
	{
		iMonth=m;
		iDay=d;
		iYear=y;
	}
	friend ostream& operator<<(ostream& os,const Date t)	//重载函数，友元函数
	{
		cout << "Month: " << t.iMonth << ' ' ;
		cout << "Day: " << t.iDay<< ' ';
		cout << "Year: " << t.iYear<< ' ' ;
		return os;
```






```
        }
        void Display()
        {
                cout << "Month: " << iMonth;
                cout << "Day: " << iDay;
                cout << "Year: " << iYear;
                cout << endl;
        }
};
template <class T,int b>                                    //类模板
class Array                                                 //声明一个类
{
        T elem[b];
public:
        Array(){}                                           //构造函数
        T& operator[] (int sub)
        {
                assert(sub>=0&& sub<b);                     //调用 assert 来进行警告处理
                return elem[sub];
        }
};
void main()
{
        Array<Date,3> dateArray;                            //用类模板定义一个数组
        Date dt1(1,2,3);                                    //用类 Data 定义对象 dt1
        Date dt2(4,5,6);                                    //用类 Data 定义对象 dt2
        Date dt3(7,8,9);                                    //用类 Data 定义对象 dt3
        dateArray[0]=dt1;                                   //用对象给数组赋值
        dateArray[1]=dt2;
        dateArray[2]=dt3;
        for(int i=0;i<3;i++)
                cout << dateArray[i] << endl;               //输出数组
        Date dt4(10,11,13);
        dateArray[3] = dt4;                                 //弹出警告
        cout << dateArray[3] << endl;
}
```

程序运行结果如图 13.7 所示。

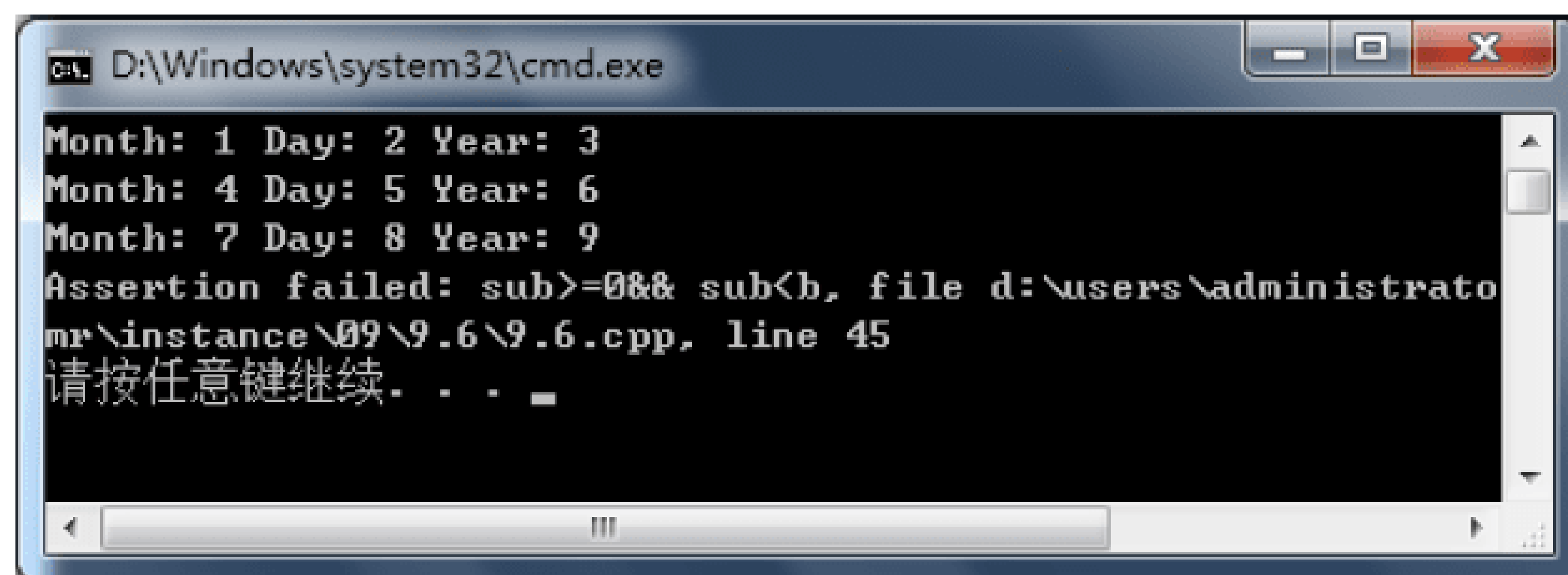


图 13.7　越界警告

程序能够及时发现 dateArray 已经越界，因为定义数组时指定数组的长度为 3，当数组下标





为 3 时说明数组中有 4 个元素，所以程序执行到 dateArray[3]时，弹出错误警告，如图 13.8 所示。

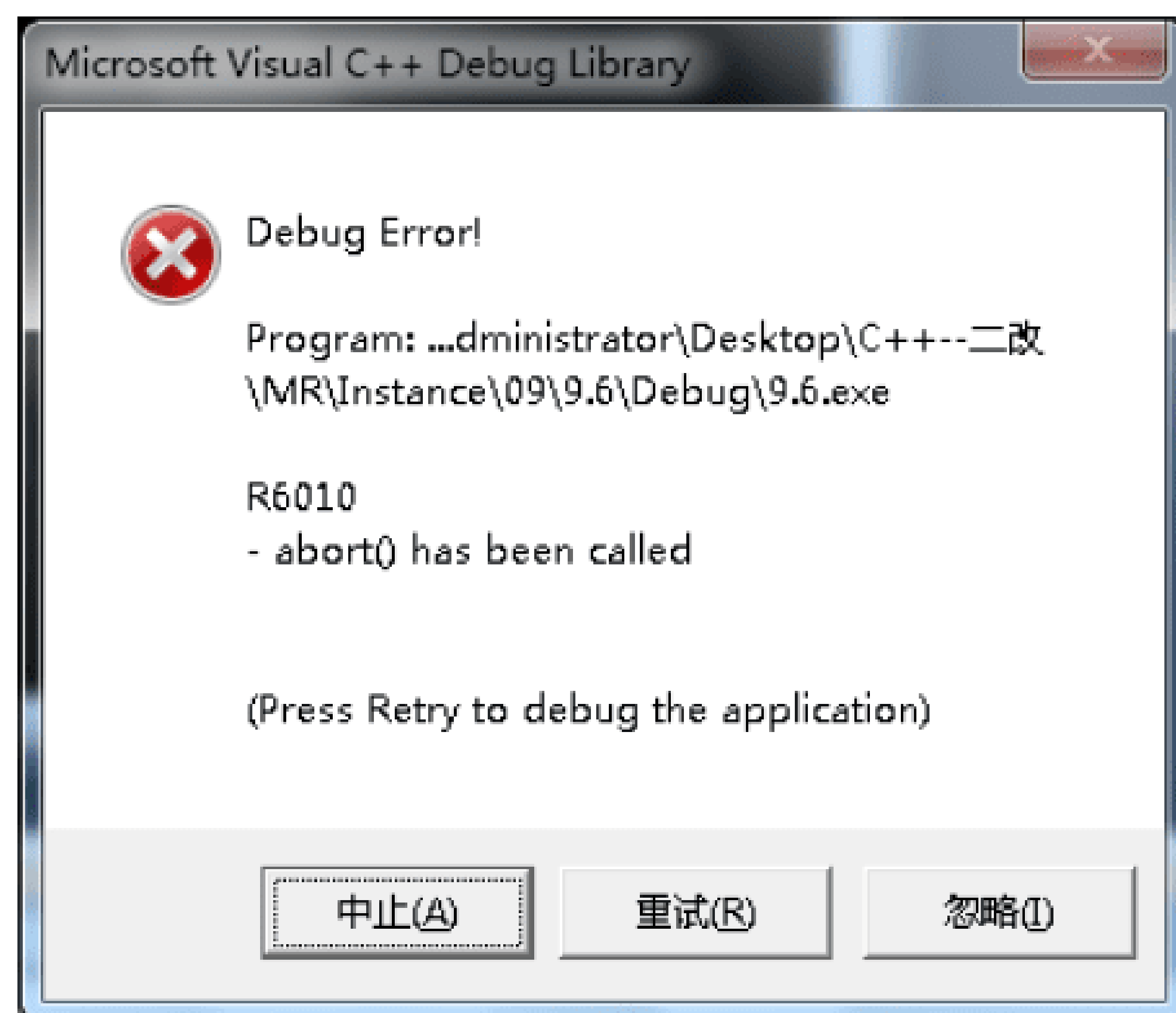


图 13.8　编译器的阻止警告

13.3　模板的使用方法

定义完模板类后如果想扩展模板新类的功能，需要对类模板进行覆盖，使模板类能够完成特殊功能。覆盖操作可以针对整个类模板、部分类模板以及类模板的成员函数。这种覆盖操作称为定制。

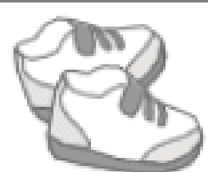
13.3.1　定制类模板

定制一个类模板，使用 template<>覆盖类模板中所定义的所有成员。

【例 13.7】 定制类模板。

实例位置：光盘\MR\Instance\13\13.7

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class Date
{
	int iMonth,iDay,iYear;
	char Format[128];
public:
	Date(int m=0,int d=0,int y=0)
	{
		iMonth=m;
		iDay=d;
		iYear=y;
	}
	friend ostream& operator<<(ostream& os,const Date t)
```





Note

```
    {
        cout << "Month: " << t.iMonth << ' ' ;
        cout << "Day: " << t.iDay<< ' ';
        cout << "Year: " << t.iYear<< ' ' ;
        return os;
    }
    void Display()
    {
        cout << "Month: " << iMonth;
        cout << "Day: " << iDay;
        cout << "Year: " << iYear;
        cout << endl;
    }
};
template <class T>
class Set
{
    T t;
public:
    Set(T st) : t(st) {}
    void Display()
    {
        cout << t << endl;
    }
};
template<>                                    //显示专用化
class Set<Date>
{
    Date t;
public:
    Set(Date st): t(st){}
    void Display()
    {
        cout << "Date :" << t << endl;
    }
};
void main()
{
    Set<int> intset(123);
    Set<Date> dt =Date(1,2,3);
    intset.Display();
    dt.Display();
}
```

程序运行结果如图 13.9 所示。

程序中定义了 Set 类模板，该模板中有一个构造函数和一个 Display 成员函数。Display 成员函数负责输出成员的值。使用类 Date 定制了整个类模板，也就是说模板类中构造函数中的参数是 Date 对象，Display 成员函数输出的也是 Date 对象。定制类模板相当于实例化一个模板类。



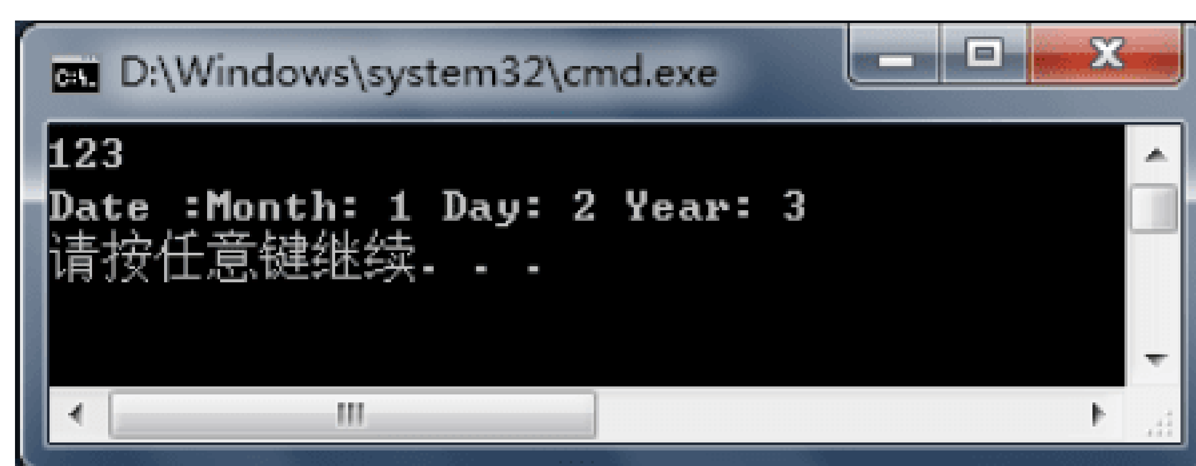


图 13.9　定制类模板



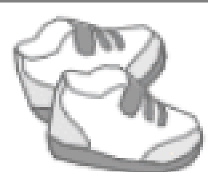
13.3.2　定制类模板成员函数

定制一个类模板，然后覆盖类模板中指定的成员。

【例 13.8】 定制类模板成员函数。

实例位置：光盘\MR\Instance\13\13.8

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class Date
{
	int iMonth,iDay,iYear;
	char Format[128];
public:
	Date(int m=0,int d=0,int y=0)
	{
		iMonth=m;
		iDay=d;
		iYear=y;
	}
	friend ostream& operator<<(ostream& os,const Date t)
	{
		cout << "Month: " << t.iMonth << ' ' ;
		cout << "Day: " << t.iDay<< ' ';
		cout << "Year: " << t.iYear<< ' ' ;
		return os;
	}
	void Display()
	{
		cout << "Month: " << iMonth;
		cout << "Day: " << iDay;
		cout << "Year: " << iYear;
		cout << std::endl;
	}
};
template <class T>
class Set
{
	T t;
public:
```



```
        Set(T st) : t(st) { }
        void Display();
};
template <class T>
void Set<T>::Display()
{
        cout << t << endl;
}
void Set<Date>::Display()
{
        cout << "Date: " << t << endl;
}
void main()
{
        Set<int> intset(123);
        Set<Date> dt =Date(1,2,3);
        intset.Display();
        dt.Display();
}
```



Note

程序运行结果如图 13.10 所示。

程序中定义了 Set 类模板，该模板中有一个构造函数和一个 Display 成员函数。程序对模板类中的 Display 函数进行覆盖，使其参数类型设置为 Date 类，这样在使用 Display 函数输出时就会调用 Date 类中的 Display 函数进行输出。

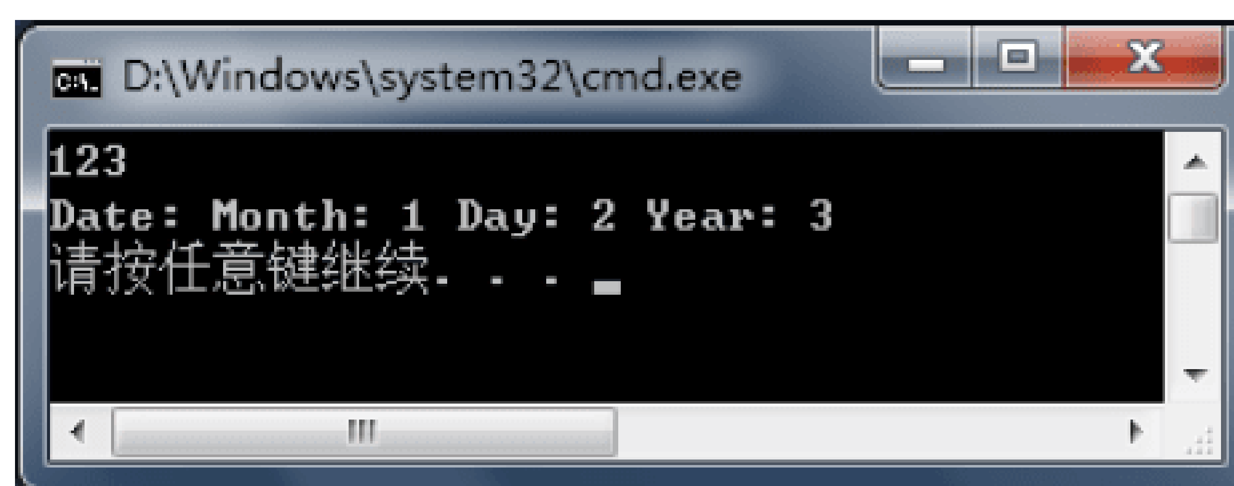


图 13.10　定制类模板成员函数

13.3.3　部分定制模板

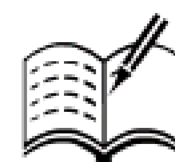
定制一个类模板，然后覆盖类模板类型参数表中的一个参数。

【例 13.9】　模板部分定制。

实例位置：光盘\MR\Instance\13\13.9

```
#include "stdafx.h"
#include <iostream>
using namespace std;
template <class T1,class T2>
class MyTemplate
{
        T1 obj1;
        T2 obj2;
public:
        MyTemplate(T1 o1,T2 o2) : obj1(o1) ,obj2(o2){}
        void display()
        {
```





```
            cout << "Object Display" << endl;
            cout << "Object 1:" << obj1 << endl;
            cout << "Object 2:" << obj2 <<endl;
            cout << endl;
        }
};
template <class T>
class MyTemplate<T, char>
{
        T obj1,obj2;
public:
        MyTemplate(T o1,char c) : obj1(o1) ,obj2(o1)
        {obj2+=(int)c;}
        void display()
        {
            cout << "Object Display" << endl;
            cout << "Object 1:" << obj1 << endl;
            cout << "Object 2:" << obj2 <<endl;
            cout << endl;
        }
};
int main()
{
        MyTemplate<int,int>mt1(10,20);
        MyTemplate<int,int>mt2(10,'b');
        mt1.display();
        mt2.display();
        return 1;
}
```

程序运行结果如图 13.11 所示。

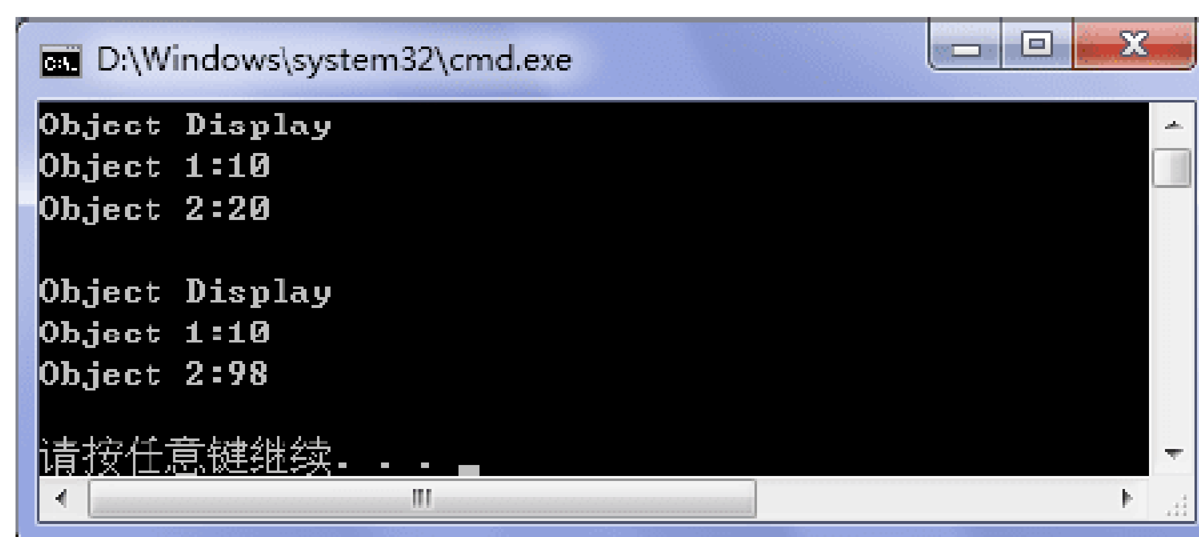
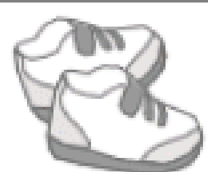


图 13.11　模板部分定制

程序中的 MyTemplate 类模板的一个参数被覆盖为 char，在模板类 MyTemplate 的构造函数中，用第一个参数的对象和 char 值相加。如果第一个参数被设置为 int 类型，那么 char 可以转换为 int 类型，完成和第一个参数的例值的相加。

13.4　链表类模板

链表是一种常用的数据结构，创建链表类模板就是创建一个对象的容器，在容器内可以对不同类型的对象进行插入、删除和排序等操作。C++标准模板中有链表类模板，本节将主要实现简单的链表类模板。





Note

13.4.1 建立单向链表

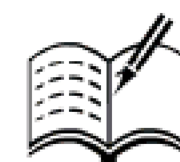
在介绍类模板之前，先来设计一个简单的单向链表。链表的功能包括向尾节点添加数据、遍历链表中的节点和在链表结束时释放所有节点。例如定义一个链表类。

【例 13.10】 简单链表的实现。

实例位置：光盘\MR\Instance\13\13.10

```
#include "stdafx.h"
class CNode                                         //定义一个节点类
{
public:
    CNode *m_pNext;                                 //定义一个节点指针，指向下一个节点
    int    m_Data;                                  //定义节点的数据
    CNode()                                         //定义节点类的构造函数
    {
        m_pNext = NULL;                             //将 m_pNext 设置为空
    }
};
class CList                                         //定义链表类 CList 类
{
private:
    CNode *m_pHeader;                               //定义头节点
    int    m_NodeSum;                               //节点数量
public:
    CList()                                         //定义链表的构造函数
    {
        m_pHeader = NULL;                           //初始化 m_pHeader
        m_NodeSum = 0;                              //初始化 m_NodeSum
    }
    CNode* MoveTrail()                              //移动到尾节点
    {
        CNode* pTmp = m_pHeader;                    //定义一个临时节点，将其指向头节点
        for (int i=1;i<m_NodeSum;i++)               //遍历节点
        {
            pTmp = pTmp->m_pNext;                   //获取下一个节点
        }
        return pTmp;                                //返回尾节点
    }
    void AddNode(CNode *pNode)                      //添加节点
    {
        if (m_NodeSum == 0)                         //判断链表是否为空
        {
            m_pHeader = pNode;                      //将节点添加到头节点中
        }
        else                                        //链表不为空
        {
            CNode* pTrail = MoveTrail();            //搜索尾节点
```



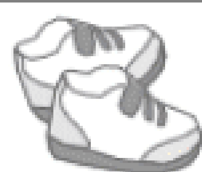


```
                pTrail->m_pNext = pNode;                  //在尾节点处添加节点
            }
            m_NodeSum++;                                  //使链表节点数量加 1
        }
        void PassList()                                   //遍历链表
        {
            if (m_NodeSum > 0)                            //判断链表是否为空
            {
                CNode* pTmp = m_pHeader;                  //定义一个临时节点，将其指向头节点
                printf("%4d",pTmp->m_Data);               //输出节点数据
                for (int i=1;i<m_NodeSum;i++)             //遍历其他节点
                {
                    pTmp = pTmp->m_pNext;                 //获取下一个节点
                    printf("%4d",pTmp->m_Data);           //输出节点数据
                }
            }
        }
        ~CList()                                          //定义链表析构函数
        {
            if (m_NodeSum > 0)                            //链表不为空
            {
                CNode *pDelete = m_pHeader;               //定义一个临时节点，指向头节点
                CNode *pTmp = NULL;                       //定义一个临时节点
                for(int i=0; i< m_NodeSum; i++)           //遍历节点
                {
                    pTmp = pDelete->m_pNext;              //获取下一个节点
                    delete pDelete;                       //释放当前节点
                    pDelete = pTmp;                       //将下一个节点设置为当前节点
                }
                m_NodeSum = 0;                            //将 m_NodeSum 设置为 0
                pDelete = NULL;                           //将 pDelete 设置为空
                pTmp = NULL;                              //将 pTmp 设置为空
            }
            m_pHeader = NULL;                             //将 m_pHeader 设置为空
        }
};
```

链表类 CList 以 CNode 作为元素，通过 MoveTrail 成员函数将链表指针移动到末尾，通过 AddNode 成员函数添加一个节点。

声明一个链表对象，向其中添加节点，并遍历链表节点。代码如下：

```
int main(int argc, char* argv[])
{
        CList list;                                       //定义链表对象
        for(int i=0; i<5; i++)                            //利用循环向链表中添加 5 个节点
        {
            CNode *pNode = new CNode();                   //构造节点对象
            pNode->m_Data = i;                            //设置节点数据
            list.AddNode(pNode);                          //添加节点到链表
```



```
    }
    list.PassList();                              //遍历节点
    cout << endl;                                 //输出换行
    return 0;
}
```



Note

程序运行结果如图 13.12 所示。

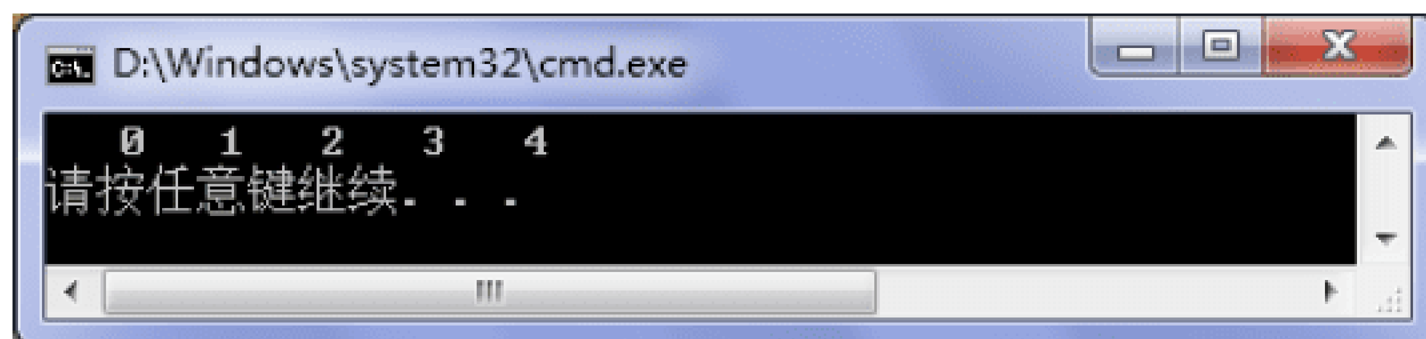


图 13.12　简单链表

程序向链表中添加了 5 个元素，然后调用 PassList 成员函数完成对链表元素的遍历。

13.4.2　链表类模板的使用

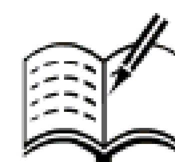
链表类 Clist 的一个最大缺陷就是链表不够灵活，其节点只能是 CNode 类型。让 CList 能够适应各种类型的节点的最简单方法就是使用类模板。类模板的定义与函数模板类似，以关键字 template 开始，其后是由尖括号<>构成的模板参数。下面重新修改链表类 CList，以类模板的形式进行改写。

【例 13.11】 使用 CList 类模板。

实例位置：光盘\MR\Instance\13\13.11

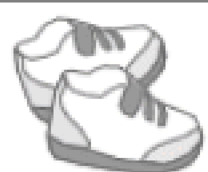
```
template <class Type>                             //定义类模板
class CList                                       //定义 CList 类
{
private:
    Type *m_pHeader;                              //定义头节点
    int    m_NodeSum;                             //节点数量
public:
    CList()                                       //定义构造函数
    {
        m_pHeader = NULL;                         //将 m_pHeader 置为空
        m_NodeSum = 0;                            //将 m_NodeSum 置为 0
    }
    Type* MoveTrail()                             //获取尾节点
    {
        Type *pTmp = m_pHeader;                   //定义一个临时节点，将其指向头节点
        for (int i=1;i<m_NodeSum;i++)             //遍历链表
        {
            pTmp = pTmp->m_pNext;                 //将下一个节点指向当前节点
        }
        return pTmp;                              //返回尾节点
    }
    void AddNode(Type *pNode)                     //添加节点
```





```
    {
        if (m_NodeSum == 0)                         //判断链表是否为空
        {
            m_pHeader = pNode;                      //在头节点处添加节点
        }
        else                                        //链表不为空
        {
            Type* pTrail = MoveTrail();             //获取尾节点
            pTrail->m_pNext = pNode;                //在尾节点处添加节点
        }
        m_NodeSum++;                                //使节点数量加 1
    }
    void PassList()                                 //遍历链表
    {
        if (m_NodeSum > 0)                          //判断链表是否为空
        {
            Type* pTmp = m_pHeader;                 //定义一个临时节点，将其指向头节点
            printf("%4d",pTmp->m_Data);             //输出头节点数据
            for (int i=1;i<m_NodeSum;i++)           //利用循环访问节点
            {
                pTmp = pTmp->m_pNext;               //获取下一个节点
                printf("%4d",pTmp->m_Data);         //输出节点数据
            }
        }
    }
    ~CList()                                        //定义析构函数
    {
        if (m_NodeSum > 0)                          //判断链表是否为空
        {
            Type *pDelete = m_pHeader;              //定义一个临时节点，将其指向头节点
            Type *pTmp = NULL;                      //定义一个临时节点
            for(int i=0; i< m_NodeSum; i++)         //利用循环遍历所有节点
            {
                pTmp = pDelete->m_pNext;            //将下一个节点指向当前节点
                delete pDelete;                     //释放当前节点
                pDelete = pTmp;                     //将当前节点指向下一个节点
            }
            m_NodeSum = 0;                          //设置节点数量为 0
            pDelete = NULL;                         //将 pDelete 置为空
            pTmp = NULL;                            //将 pTmp 置为空
        }
        m_pHeader = NULL;                           //将 m_pHeader 置为空
    }
};
```

上述代码利用类模板对链表类 CList 进行了修改，实际上是在原来链表的基础上将链表中出现 CNode 类型的地方替换为模板参数 Type。下面再定义一个节点类 CNet，演示模板类 CList 是如何适应不同的节点类型的。代码如下：






```
class CNet                                      //定义一个节点类
{
public:
        CNet *m_pNext;                          //定义一个节点类指针
        char    m_Data;                         //定义节点类的数据成员
        CNet()                                  //定义构造函数
        {
                m_pNext = NULL;                 //将 m_pNext 置为空
        }
};
int main(int argc, char* argv[])
{
        CList<CNode> nodelist;                  //构造一个类模板例
        for(int n=0; n<5; n++)                  //利用循环向链表中添加节点
        {
                CNode *pNode = new CNode();     //创建节点对象
                pNode->m_Data = n;              //设置节点数据
                nodelist.AddNode(pNode);        //向链表中添加节点
        }
        nodelist.PassList();                    //遍历链表
        cout <<endl;                            //输出换行
        CList<CNet> netlist;                    //构造一个类模板例
        for(int i=0; i<5; i++)                  //利用循环向链表中添加节点
        {
                CNet *pNode = new CNet();       //创建节点对象
                pNode->m_Data = 97+i;           //设置节点数据
                netlist.AddNode(pNode);         //向链表中添加节点
        }
        netlist.PassList();                     //遍历链表
        cout << endl;                           //输出换行
        return 0;
}
```

程序运行结果如图 13.13 所示。

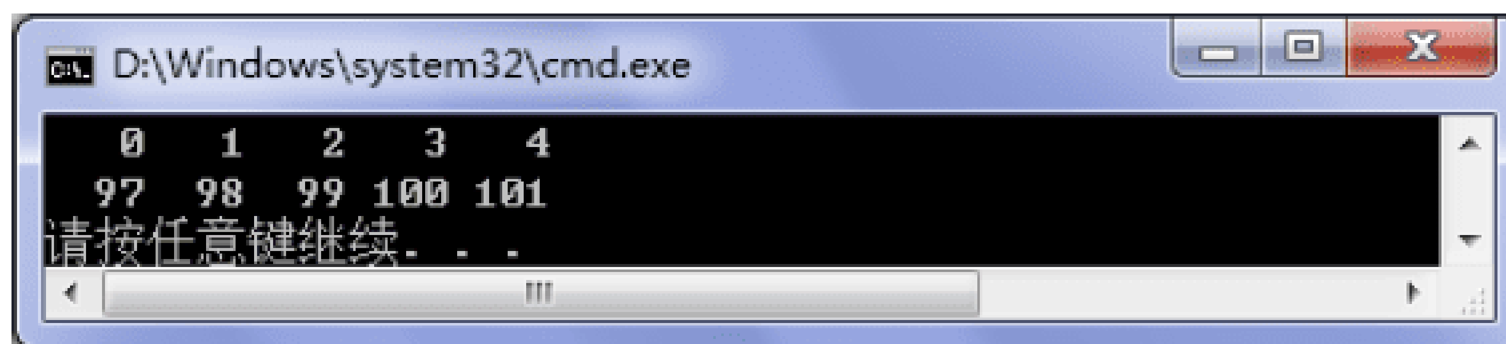


图 13.13　使用 CList 类模板

类模板 CList 虽然能够使用不同类型的节点，但对节点的类型是有一定要求的：一是节点类必须包含一个指向自身的指针类型成员 m_pNext，因为在 CList 中访问了 m_pNext 成员；二是节点类中必须包含数据成员 m_Data，其类型被限制为数字类型或有序类型。

13.4.3　类模板的静态数据成员

在类模板中可以定义静态的数据成员，类模板中的每个例都有自己的静态数据成员，而不是



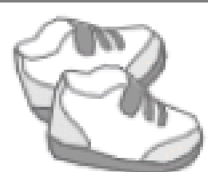


所有的类模板例共享静态数据成员。

【例 13.12】 在类模板中使用静态数据成员。

实例位置：光盘\MR\Instance\13\13.12

```
#include "stdafx.h"
#include <iostream>
using namespace std;
template <class Type>
class CList                                    //定义 CList 类
{
private:
	Type *m_pHeader;
	int    m_NodeSum;
public:
	static int m_ListValue;                //定义静态数据成员
	CList()
	{
		m_pHeader = NULL;
		m_NodeSum = 0;
	}
};
class CNode                                    //定义 CNode 类
{
public:
	CNode *m_pNext;
	int    m_Data;
	CNode()
	{
		m_pNext = NULL;
	}
};
class CNet                                     //定义 CNet 类
{
public:
	CNet *m_pNext;
	char    m_Data;
	CNet()
	{
		m_pNext = NULL;
	}
};
template <class Type>
int CList<Type>::m_ListValue = 10;            //初始化静态数据成员
int main(int argc, char* argv[])
{
	CList<CNode> nodelist;
	nodelist.m_ListValue = 2008;
	CList<CNet> netlist;
	netlist.m_ListValue = 88;
	cout<<nodelist.m_ListValue<< endl;
```



```
	cout<<netlist.m_ListValue<<endl;
	return 0;
}
```



Note

程序运行结果如图 13.14 所示。

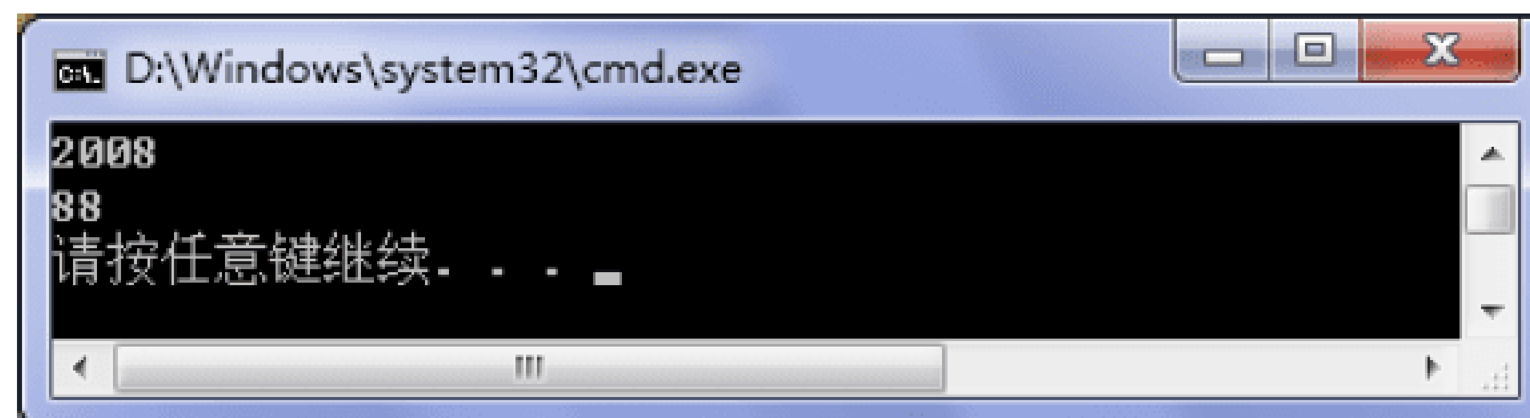


图 13.14　类模板的静态数据成员

由于模板例 nodelist 和 netlist 均有各自的静态数据成员，所以 m_ListValue 的值是不同的。但是对于同一类型的模板例，其静态数据成员是共享的。

13.5　综 合 应 用

13.5.1　除法函数模板

【例 13.13】 本例设计一个函数模板，计算两个对象相除的结果，并将结果返回。首先定义一个模板类型，再定义模板函数，制定要实现除法功能的函数。调用时给出相除的两个数的数据类型。代码如下：

实例位置：光盘\MR\Instance\13\13.13

```
#include "stdafx.h"
#include <iostream>
using namespace std;
template<class Type>                          //定义模板类型
Type Plus(Type t1,Type t2)                    //定义模板函数
{
	return t1 / t2;
}
int _tmain(int argc, _TCHAR* argv[])
{
	int a = 0, b = 0;
	cout<<"输入被除数 a：\n";
	cin>>a;
	cout<<"输入除数 b：\n";
	cin>>b;
	if(0 == b)
	{
		cout<<"除数不能为 0，请重新输入 ！\n";
		return 0;
	}
```





```
    cout<<"a 除以 b 结果是："<<Plus<int>(a,b)<<endl;
    return 0;
}
```

程序运行结果如图 13.15 所示。

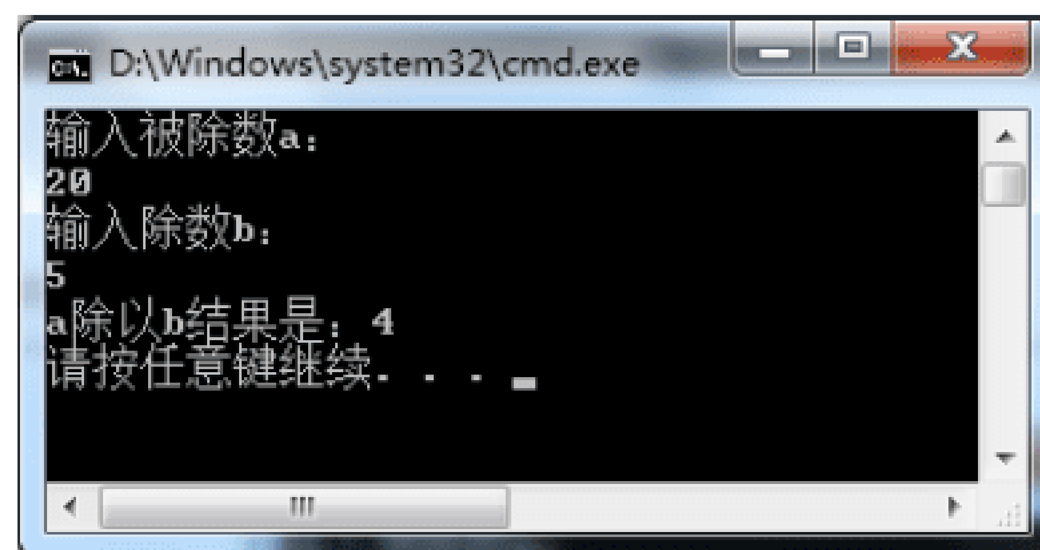


图 13.15　除法函数模板

13.5.2　取得数据间最大值

【例 13.14】 本例设计一个类模板，在类模板中声明一个成员函数，用该函数得到两个同类型的数据。然后再设计成员函数 max 得到两个数据间的最大值。定义类模板时，将模板类型作为成员函数的返回值和参数列表类型就可以实现这个目的。代码如下：

实例位置：光盘\MR\Instance\13\13.14

```
#include "stdafx.h"
#include <iostream>
#define MyType int                          //定义一个数据类型的宏
using namespace std;
template<class T>                           //定义一个模板类型
class A                                     //定义类 A
{
public:
    T getMax(T t1,T t2)                     //成员函数（调用模板定义模板函数）
    {
        if(t1 > t2)
        {
            return t1;
        }
        else
            return t2;
    }
};
int main()
{
    MyType a = 0,b = 0;
    cout<<"输入一个数：\n";
    cin>>a;
    cout<<"再输入一个数：\n";
    cin>>b;
    A<MyType> test;
    cout<<"大数是："<<test.getMax(a,b)<<endl;
    return 0;
}
```

程序运行结果如图 13.16 所示。





Note

13.5.3 不同类型数组管理

【例 13.15】 本例实现使用一个通用数组模板 CArray 管理各种类型的数组。

实现过程：定义一个类模板，带有一个可替换的参数类型 type，重载运算符[]模拟数组元素的引用。定义两个对象分别模拟 int 型数组和 Ctest 类型数组的使用。代码如下：



图 13.16 取得数据间最大值

实例位置：光盘\MR\Instance\13\13.15

```
#include "stdafx.h"
#include <iostream>
#define ARRAYSIZE 4
using namespace std;
class Ctest                                    //声明一个 Ctest 类
{
private:
	int data;
public:
	Ctest()                                //声明构造函数
	{
		data = 0;
	}
	int getdata()const                     //在类中对成员函数进行定义，得到成员变量的数据
	{
		return data;
	}
	void setdata(int m_data)               //更改成员变量
	{
		data = m_data;
	}
};
template <class type>                          //声明一个类模板，参数类型为 type
class CArray
{
private:
	type *ptemp;
public:
	CArray();
	~CArray();
	type& operator [](int offset);         //重载运算符[]
};
template <class type>
CArray<type>::CArray()                         //构造函数
{
	ptemp = new type[ARRAYSIZE];           //申请 type 类型的空间模拟数组
```





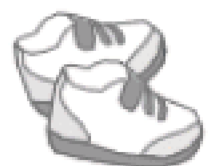
Note

```
}
template <class type>
CArray<type>::~CArray()                           //析构函数
{
    delete []ptemp;                               //释放动态申请的内存
}
template <class type>
type& CArray<type>::operator [](int offset)       //重载运算符[]
{
    return ptemp[offset];                         //返回 ptemp 指向的元素的引用
}
int _tmain(int argc, _TCHAR* argv[])
{
    CArray<int> intarray;                         //定义 int 对象
    CArray<Ctest> testarray;                      //定义 Ctest 类型对象
    for(int i = 0;i < 4;i++)
    {
        intarray[i] = i;                          //给整型数组元素赋值
        testarray[i].setdata(i*10);               //给 Ctest 型数组元素赋值
    }
    cout<<"intarray:\n";
    for(int i = 0;i < 4;i++)
    {
        cout<<intarray[i]<<endl;                  //输出 intarray 的值
    }
    cout<<"testarray:\n";
    for(int i = 0;i < 4;i++)
    {
        cout<<testarray[i].getdata()<<endl;       //输出 testarray 的值
    }
    return 0;
}
```

程序运行结果如图 13.17 所示。

```
D:\Windows\system32\cmd.exe
intarray:
0
1
2
3
testarray:
0
10
20
30
请按任意键继续. . .
```

图 13.17　通过数组模板管理各种类型的数组



13.6 本章常见错误



Note

13.6.1 函数模板与类模板的区别

函数模板的实例化是由编译程序在处理函数调用时自动完成的，而类模板的实例化必须由程序员在程序中显式地指定。

13.6.2 成员函数在类外实现时不要带默认值

如果类内声明的成员函数带有默认值，在类外实现此成员函数时，只写函数形参即可，不要把默认值也带上，否则编译出错。例如：

```
class A
{
public:
        void display(int x = 0);            //类内声明函数，带有默认值
};
//类外实现
void A::display(int x = 0)                  //错误，提示重定义
{…}
void A::display(int x)                      //正确
{…}
```

13.6.3 函数默认顺序从右向左

函数参数可以带默认值，在调用函数时如果不传递参数，函数会按照默认值执行。声明函数时，默认要从右向左依次进行，如果右边的没有默认，左边的默认了，编译会出错。

```
void dis(int a = 0,int b);                  //错
void dis(int a ,int b = 0);                 //对
```

13.7 本 章 小 结

模板是 C++的高级特性，一个模板可以定义一组函数或类，它使用数据类型和类名作为参数，建立具有类型安全的类库集合和函数集合。模板可以对作为模板参数的数据类型进行相同的操



作，大大减少了代码量，提高了代码效率，更是方便了大规模软件的开发。标准 C++库（STL）在很大程度上依赖于模板。通过本章的学习，可以使读者对 C++语言有更深入的了解。



Note

13.8　跟我上机

参考答案：光盘\MR\跟我上机

设计一个模板类，包含一个整型变量和一个 T 类型的指针，在构造函数中申请 10 个字节的空间，调用赋值函数 set 向空间内写入字符串“1234”，然后用 out 函数输出显示。实现如下：

```
#include "stdafx.h"
#include <string>
#include <iostream>
using std::cout;
using std::endl;
template <class T,int n>
class A                                         //声明一个模板类
{
	int size;
	T *p;
public:
	A();
	void set(T *str);                       //给申请的空间赋值
	void out();                             //输出字符串
};
template <class T,int n>
A<T,n>::A()
{
	size = n;
	p = new T(size);
}
template <class T,int n>
void A<T,n>::set(T *str)
{
	strcpy(p,str);
}
template <class T,int n>
void A<T,n>::out()
{
	cout<<p<<endl;
}
int _tmain(int argc, _TCHAR* argv[])
{
	A<char,10> a;
	a.set("1234");
	a.out();
	return 0;
}
```

第14章

代码整理

（视频讲解：52 分钟）

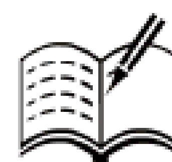
在我们阅读或使用代码时，总是希望它的可读性良好。C++提供了类型别名和枚举用来约束种类的名称。类型推导的特性则可以使复杂的空间名、类型名的数据声明得到简化。使用异常处理，程序运行时对可能发生的错误进行控制，防止系统灾难性错误的发生。

本章能够完成的主要范例（已掌握的在方框中打勾）

- [ ] 使用 typedef 给数据类型重命名
- [ ] 使用枚举类型定义变量
- [ ] 演示类型推导
- [ ] 使用自定义异常类
- [ ] 获取不同异常的 try…catch
- [ ] 使用带参数的宏实现求两个数乘积
- [ ] 使用带参数的宏求圆面积



14.1　结构体概述



Note

结构体是一种自定义数据类型。声明结构体时使用的关键字是 struct，定义一种结构体的一般形式为：

```
struct 结构体名
{
    成员表列
};
```

结构体类型与基本类型一样是从 C 语言中继承下来的。如果您是 C 语言的使用者则需要知道 C++结构体与 C 语言结构体的区别。C 语言中并没有继承、成员函数等概念，所以 C 语言中的结构体成员只能包含 C 语言中的数据类型，不能包含成员函数。

C++中的结构体和类的使用方法几乎一样。它包含 this 指针，可以继承也可以被继承。创建、销毁和复制时均调用相应的构造、析构和复制构造函数。它包含虚表，可以被抽象化等。

一般情况下，C++中经常使用类来完成面向对象的任务。在兼容 C 语言编写的源文件的情况下，有时会使用结构体。



注意：

结构体和类有两点区别：第一，结构体的默认访问权限为 public，而类中为 private；第二，结构体无法使用类模板。

14.2　重命名数据类型

C++允许使用关键字 typedef 给一个数据类型定义一个别名。例如：

```
typedef int flag;                    //给 int 数据类型取一个别名
```

这样，程序中 flag 就可以作为 int 的数据类型来使用：

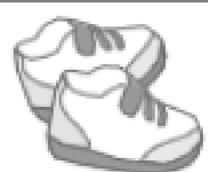
```
flag a;
```

a 实质上是 int 类型的数据，此时 int 类型的别名就是 flag。

类或者结构在声明时使用 typedef：

```
typedef class asdfghj{
    成员列表
}myClass,ClassA;
```

这样就令声明的类拥有 myClass、ClassA 两个别名。





Note

typedef 主要的用途如下：

☑ 很复杂的基本类型名称，如函数指针 int (*)(int i)：

```
typedef int (*)(int i) pFun;          //用 pFun 代替函数指针 int (*)(int i)
```

☑ 使用其他人开发的类型时，使它的类型名符合自己的代码习惯（规范）。

typedef 关键字具有作用域，范围是别名声明所在的区域（包含名称空间）。

【例 14.1】 三只宠物犬。

实例位置：光盘\MR\Instance\14\14.1

```
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;
namespace pet
{
    typedef string kind;
    typedef string petname;
    typedef string voice;
    typedef class dog
    {
        private:
            kind m_kindName;                    //宠物狗种类
        protected:                              //假如有其他需要子类继承，则不需要使用种类这个属性
            petname m_dogName;
            int m_age;
            voice m_voice;
            void setVoice(kind name);
        public:
        dog(kind name);
        void sound();
        void setName(petname name);
    }Dog,DOG;                                   //声明了别名，用 Dog，DOG 代替类 dog
    void dog::setVoice(kind name)
    {
        if(name == "北京犬")
        {
            m_voice = "嗷嗷";
        }
        else if(name == "狼犬")
        {
            m_voice = "呜嗷";
        }
        else if(name == "黄丹犬")
        {
            m_voice = "喔嗷";
        }
    }
    dog::dog(kind name)
```






```
        {
            m_kindName = name;
            m_dogName = name;
            setVoice(name);
        }
        void dog::sound()
        {
            cout<<m_dogName<<"发出"<<m_voice<<"的叫声"<<endl;
        }
        void dog::setName(petname name)
        {
            m_dogName = name;
        }
}
using pet::dog;                                //使用 pet 空间的宠物犬 dog 类
using pet::DOG;
int main()
{
        dog a = dog("北京犬");                  //名称空间的类被包含进来后，可以直接使用
        pet::Dog b = pet::Dog("狼犬");          //别名仍需要使用名字空间
        pet::DOG c = pet::DOG("黄丹犬");
        a.setName("小白");
        c.setName("阿黄");
        a.sound();
        b.sound();
        c.sound();
        return 0;
}
```

程序运行结果如图 14.1 所示。



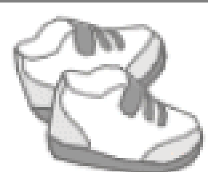
图 14.1　执行结果

在 pet 名称空间中定义了多种类型别名。这些别名的实际类型不发生改变，在主函数内演示了如何使用名称空间中的类别名。

宠物狗 dog 类中使用 string 类来区分小狗的种类，通过 setVoice 函数设定每种小狗的声音。那么，有没有比使用 string 对象更轻便的办法呢？除了建立 3 个子类之外有没有更简便一些的方法呢？在 14.3 节我们将继续讨论。

14.3　枚举类型的应用

在事物的概念中，有些数据只需要分出类别作为标识，使用整型数据 int 可以做到这一点。但对于编程者或者代码阅读者而言，很难将一群不直观的数字与概念联系起来。以 14.2 节的宠物犬 dog 类为例，建立一个 int 型成员变量，当它的值为 0 时代表北京犬，值为 1 时代表狼犬，值为 2 时代表黄丹犬。这样执行效率会更高一些，但是很难将 0、1、2 这些数字的值与犬的种类





相关联起来。

C++提供了枚举类型 enum，它的声明形式如下：

```
enum  枚举的名称{枚举 1，枚举 2，枚举 3… 枚举 n…};
```

枚举代表了事物概念的分类。使用枚举类型数据的形式如下：

```
枚举类型  变量名=枚举 n;
```

也就是说，枚举类型的名称作为数据类型来使用，它的值可以为定义的枚举其中之一。

在改进 14.2 节的宠物犬类之前，有必要了解一下枚举类型的实质。在程序中定义一个星期的枚举类型：

```
enum week{Monday,Tuesday,Wednesday,Thursday,Friday,Seturday,Sunday};
```

枚举的作用域和它的声明位置相对应，以如下方式使用它（假设程序包含标准输出流）：

```
week k = k3;                                    //定义一个枚举变量 k，赋值 k3
if(Monday == 0)
{
	cout<<Monday<<endl;
}
if(Wednesday == 2)
{
	cout<<k<<endl;
}
```

两个 cout 会被依次执行，输出的结果为 0 和 2。原来枚举类型定义的枚举实质上是个从 0 开始，递增 1 的常量整数数列，它将字面值包装到了标识符中。在编写程序中最好依然按照枚举所定义的标识符使用，这样才能保持代码的直观性。

在定义枚举类型时，也可以给枚举列表中的枚举元素初始化，被初始化的元素后面的元素依次比它大 1。例如，“enum xx{a,b=4,c,d,e};”定义一个枚举类型 xx，b 默认是 1，现在初始化为 4，结果 a 为 0，b 为 4，c 为 5，后面依此类推。



注意：

枚举中各项的名称不能和关键字、数据名、其他枚举的项等相冲突。

下面修改例 14.1 中的宠物狗 dog 类。

【例 14.2】 宠物狗的英文称呼。

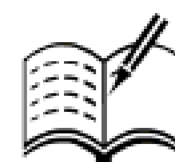
实例位置：光盘\MR\Instance\14\14.2

本例将 dog 类的声明和实现分离。

pet.h 声明了 dog 类和 pet 名称空间：

```
#include <string>
using    std::string;
```



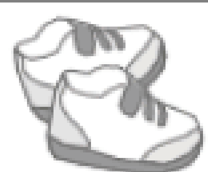



```
enum Edog{PeiKingese,demi_wolf,Huangdan};           //英文名字枚举
enum Cdog{JingBa,LangGou,HuangDan};                 //拼音名字枚举，HuangDan 的定义避免了命名冲突
namespace pet
{
	//typedef string kind;                          //换为枚举类型
	typedef string petname;
	typedef string voice;
	typedef class dog
	{
		private:
			Cdog m_kindName;                        //拼音宠物狗枚举种类
		protected:                                  //假如有别的子类需要继承，则不需要使用这个属性
			petname m_dogName;
			int m_age;
			voice m_voice;
			void setVoice(Cdog name);               //从传递 string 类型变成传递整型数据
			void setDefaultName(Cdog name);         //设置默认名字
		public:
			dog(Cdog name);                         //从传递 string 类型变成传递整型数据
			void sound();
			void setName(petname name);
			string getName();
	}Dog,DOG;                                       //声明了别名
}
```

在本实例中定义了两个枚举类型。

dog.cpp 完成了 dog 类的实现，switch 语句支持枚举类型：

```
#include "stdafx.h"
#include "pet.h"
#include <iostream>
using std::cout;
using std::endl;
using namespace pet;
void dog::setVoice(Cdog name)
{
	switch(name){
	case JingBa:
		m_voice = "嗷嗷";
		break;
	case LangGou:
		m_voice = "呜嗷";
		break;
	case HuangDan:
		m_voice = "喔嗷";
		break;
	 default:
		m_voice = "-----";
	}
}
```





Note

```
void dog::setDefaultName(Cdog name)
{
	switch(name){
	case JingBa:
		m_dogName = "京巴";
		break;
	case LangGou:
		m_dogName = "狼狗";
		break;
	case HuangDan:
		m_dogName = "黄丹";
		break;
	default:
		m_dogName = "迷之犬";
	}
}
dog::dog(Cdog name)
{
	m_kindName = name;
	setDefaultName(name);
	setVoice(name);
}
void dog::sound()
{
	cout<<m_dogName<<"发出"<<m_voice<<"的叫声"<<endl;
}
void dog::setName(petname name)
{
	m_dogName = name;
}
string dog::getName()
{
	return m_dogName;
}
```

main.cpp 程序的入口：

```
#include "stdafx.h"
#include <iostream>
#include "pet.h"
using std::cout;
using std::endl;
using pet::dog;                                  //使用 pet 空间的宠物犬 dog 类
using pet::DOG;
int main()
{
	cout<<"我领养了 2 只小狗。"<<endl;
	dog myDog1 = dog(JingBa);                //名称空间的类被包含进来后，可以直接使用
	pet::Dog myDog2= pet::Dog(LangGou);      //别名仍需要使用名字空间
```





Note

```
        myDog2.setName("小黑");
        cout<<"小狗们发出叫声："<<endl;
        myDog1.sound();
        myDog2.sound();
        cout<<"一个外国人领养了 4 只小狗"<<endl;
        //dog dog1 = dog(PeiKingese); //出现类型转换问题，虽然字面值相同，但无法隐式转换
        dog dog1 = dog((Cdog)PeiKingese);
        dog dog2 = dog((Cdog)demi_wolf);
        dog dog3 = dog((Cdog)HuangDan);        //中国人和外国人都把黄丹犬称为"huangdan"
        dog dog4 = dog((Cdog)43);              //43 明显超出枚举的范围，观察执行结果
        cout<<"3 只小狗有了英文名字"<<endl;
        dog1.setName("LuckyBoy");
        dog2.setName("Andy");
        dog3.setName("BigBow");
        cout<<"小狗们发出叫声："<<endl;
        dog1.sound();
        cout<<"唔，原来"<<dog1.getName()<<"是一只京巴"<<endl;
        dog2.sound();
        cout<<"哦，原来"<<dog2.getName()<<"是一只狼狗"<<endl;
        dog3.sound();
        cout<<"啊，原来"<<dog3.getName()<<"是一只黄丹"<<endl;
        dog4.sound();
        cout<<"嗯?请问这是什么狗?"<<endl;
        return 0;
}
```

程序运行结果如图 14.2 所示。

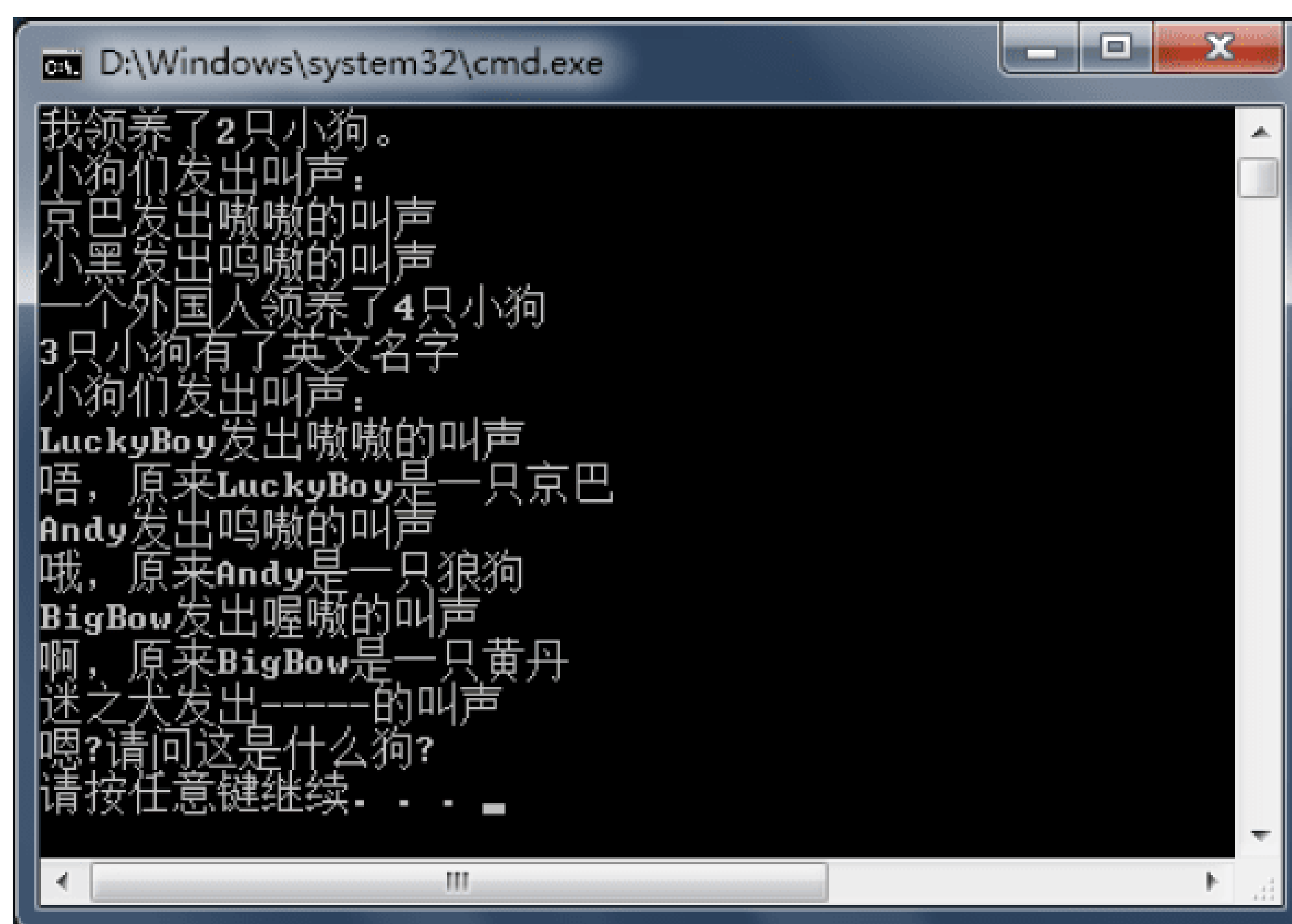
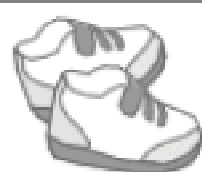
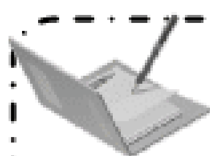


图 14.2　一样的小狗，不同的称呼

向宠物狗 dog1、dog2、dog3、dog4 的构造函数中传递了强制转换为 Cdog 的枚举。由于 Edog 枚举和 Cdog 枚举一一对应，所以程序中的 dog1～dog3 仍然是“京巴”、“狼狗”和“黄丹”。dog4 的构造函数传递了一个超出枚举范围的整数，在类中的 switch 语句中仍然可以执行。





说明：

typedef 和 enum 的作用相比较，typedef 是将数据类型的名称直接包装成另外一个命名。枚举类型是一种能够隐形转换为 int 型的数据，enum 则将 int 常量的字面值包装成了字符代码。



Note

14.4 类型推导

类型推导是 C++11 支持的一种新的特性，它对数据类型的声明具有很大的帮助。很难确认某函数在一定条件下的返回值类型，因为它可能使用了函数模板，也或者是使用类模板的对象的成员函数。C++03 标准的 auto 是一个用来标识数据自动储存方式的关键字，C++11 赋予了它新的功能：

```
int k1 = 3;
auto k2 = k1;
```

变量 k2 的类型被推导为 int 类型。同样地，使用 auto 关键字声明的数据可以被任何非空类型的数据、表达式和函数初始化：

```
auto i = func(参数列表);                //func 返回值为非空
```

函数的返回值可以使用 auto 做返回类型的声明，这就是 C++11 提供的新特性——类型推导。

decltype 也是实现这一特性的关键字。它的作用是可以获得某一表达式、函数或者数据的数据类型。

它的使用方法如下：

```
Type k = somevalue;                   //k 的数据类型为 Type，somevalue 表示的是这一类型合法的值
decltype(k) p =somevalue;             //p 被初始化为 k 的类型 Type
```

decltype(k)可以视作数据类型 Type，除了使用 decltype 初始化数据的用途之外，还可以显式转换某些数据（例如空类型指针）、向模板中传递类型等。



注意：

使用 auto 关键字声明的数据必须初始化。auto 不能作为数组的类型声明，也不可以在形参列表中使用。

【例 14.3】 演示类型推导。

实例位置：光盘\MR\Instance\14\14.3

```
#include "stdafx.h"
#include <string>
#include <iostream>
using std::cout;
```





```
using std::endl;
using std::string;
using std::string;
class human
{
        private:
        int m_nSpeed;
        string m_Name;
public:
        human(string name)
        {
                m_Name = name;
        }
        void sayHello()
        {
                cout<<"你好!我是"<<m_Name<<endl;
        }
};
int main()
{
        auto h1 = human("Mike");
        decltype(h1) h2 = human("老刘");
        h1.sayHello();
        h2.sayHello();
}
```

由 auto 声明的 h1 的类型就是初始化的 human 类。h2 的类型使用 decltype 关键字对 h1 的类型进行推导，所以也是 human 类型。程序运行结果如图 14.3 所示。

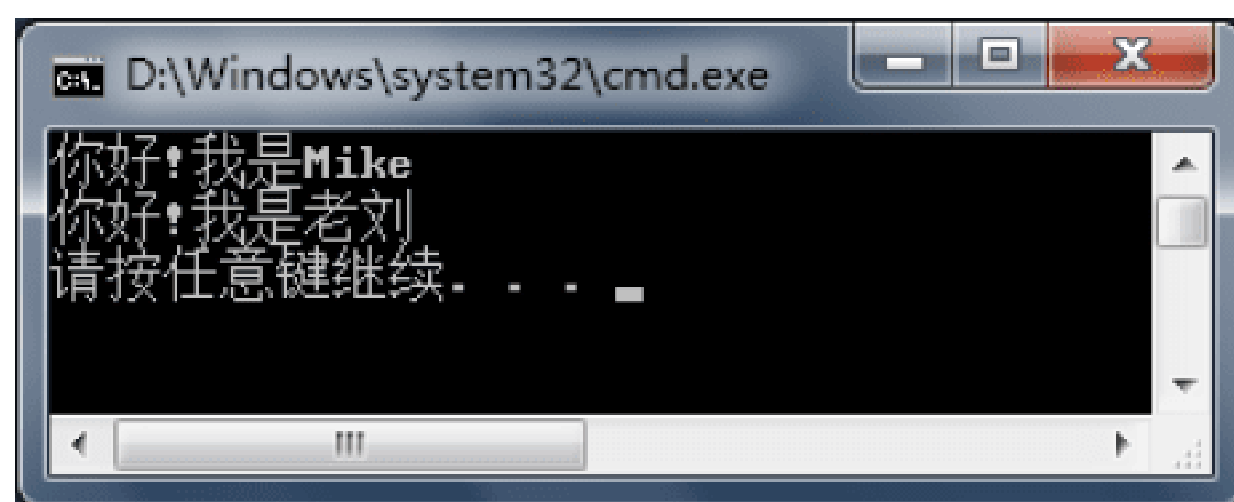
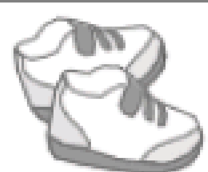


图 14.3 类型推导

14.5 异 常 处 理

异常处理是程序设计中除调试之外的另一种错误处理方法，它往往被大多数程序设计人员在实际设计中忽略。异常处理引起的代码膨胀将不可避免地增加程序阅读的困难，这对于程序设计人员来说是十分烦恼的。异常处理与真正的错误处理有一定区别，异常处理不但可以对系统错误做出反应，还可以对人为制造的错误做出反应并处理。本章将向读者介绍 C++语言对于异常处理的使用方法。



14.5.1 抛出异常



当程序执行到某一函数或方法内部时，程序本身出现了一些异常，但这些异常并不能由系统所捕获，这时就可以创建一个错误信息，再由系统捕获该错误信息并处理。创建错误信息并发送这一过程就是抛出异常。

最初异常信息的抛出只是定义一些常量，这些常量通常是整型值或字符串信息。下面代码是通过整型值创建的异常抛出：

```
#include "stdafx.h"
#include <iostream>
int main(int argc, char* argv[])
{
	try
	{
		throw 1;                    //抛出异常
	}
	catch(int error)
	{
		if (error == 1)             //异常信息
			cout << "产生异常" << endl;
	}
	return 0;
}
```

在 C++中，异常的抛出是使用 throw 关键字来实现的，在这个关键字的后面可以跟随任何类型的值。在上面的代码中将整型值 1 作为异常信息抛出，当异常捕获时就可以根据该信息进行异常的处理。

异常的抛出还可以使用字符串作为异常信息进行发送，代码如下：

```
#include "stdafx.h"
#include <iostream>
int main(int argc, char* argv[])
{
	try
	{
		throw "异常产生！";          //抛出异常
	}
	catch(char * error)             //捕获异常
	{
		cout << error << endl;
	}
	return 0;
}
```

可以看到，字符串形式的异常信息适合于异常信息的显示，但并不适合于异常信息的处理。



那么是否可以将整型信息与字符串信息结合起来作为异常信息进行抛出呢？之前说过，throw 关键字后面跟随的是类型值，所以不但可以跟随基本数据类型的值，还可以跟随类类型的值，这就可以通过类的构造函数将整型值与字符串结合在一起，并且还可以同时应用更加灵活的功能。



例如，将错误 ID 和错误信息以类对象的形式进行异常抛出。

【例 14.4】 使用自定义异常类。

实例位置：光盘\MR\Instance\14\14.4

```
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;
class CCustomError                                  //异常类
{
private:
    int m_ErrorID;                                  //异常 ID
    char m_Error[255];                              //异常信息
public:
    CCustomError()                                  //构造函数
    {
        m_ErrorID = 1;
        strcpy(m_Error,"出现异常！");
    }
    int GetErrorID(){ return m_ErrorID; }           //获取异常 ID
    char * GetError(){ return m_Error; }            //获取异常信息
};
int main(int argc, char* argv[])
{
    try
    {
        throw (new CCustomError());                 //抛出异常
    }
    catch(CCustomError* error)
    {
        //输出异常信息
        cout << "异常 ID：" << error->GetErrorID() << endl;
        cout << "异常信息：" << error->GetError() << endl;
    }
    return 0;
}
```

程序运行结果如图 14.4 所示。

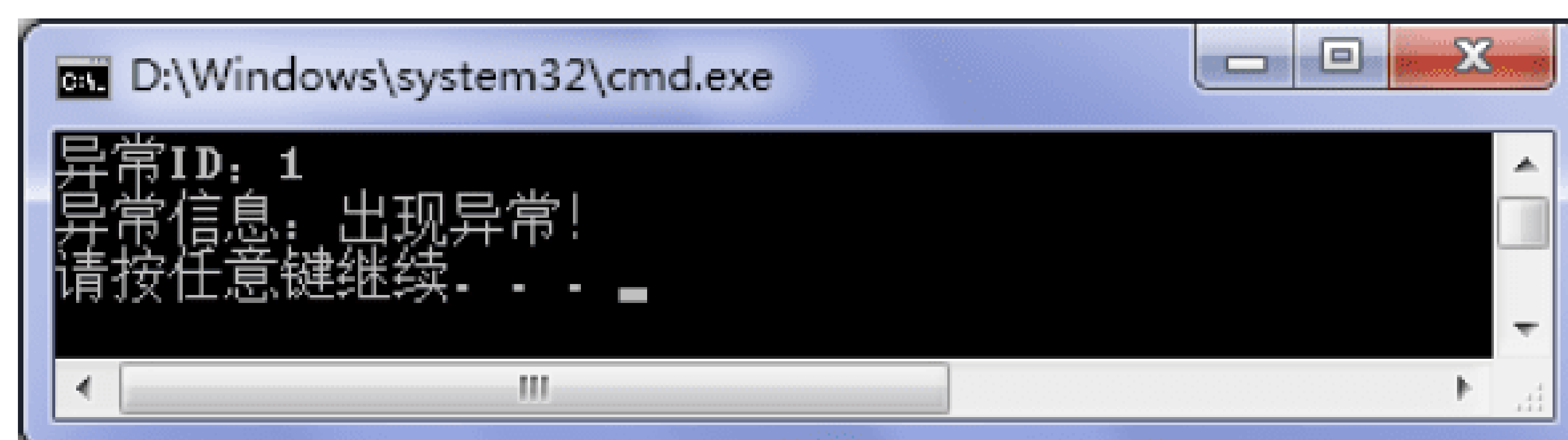
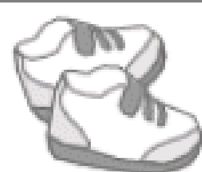


图 14.4　使用自定义异常类



代码中定义了一个异常类，这个类包含了两个内容，一个是异常 ID，也就是异常信息的编号；另一个是异常信息，也就是异常的说明文本。通过 throw 关键字抛出异常时，需要指定这两个参数。



Note

14.5.2　捕获异常

异常捕获是指当一个异常被抛出时，不一定就在异常抛出的位置来处理这个异常，而是可以在别的地方通过捕获这个异常信息后再进行处理。这样不仅增加了程序结构的灵活性，也提高了异常处理的方便性。

如果在函数内抛出一个异常（或在函数调用时抛出一个异常），将在异常抛出时退出函数。如果不想在异常抛出时退出函数，可在函数内创建一个特殊块用于解决实际程序中的问题。这个特殊块由 try 关键字组成，例如：

```
try
{
	//抛出异常
}
```

异常抛出信号发出后，一旦被异常处理器接收到就被销毁。异常处理器应具备接收任何异常的能力。异常处理器紧随 try 块之后，处理的方法由关键字 catch 引导。

```
Try
{
	...
}
catch(type obj)
{
	...
}
```

异常处理部分必须直接放在测试块之后。如果一个异常信号被抛出，异常处理器中第一个参数与异常抛出对象相匹配的函数将捕获该异常信号，然后进入相应的 catch 语句，执行异常处理程序。catch 语句与 switch 语句不同，它不需要在每个 case 语句后加入 break 去中断后面程序的执行。

下面通过 try…catch 语句来捕获一个异常。代码如下：

```
#include "stdafx.h"
#include <iostream>
#include <string>
using namespace std;
class CcustomError                                    //异常类
{
private:
	int m_ErrorID;                                    //异常 ID
	char m_Error[255];                                //异常信息
```





```
public:
	CCustomError()                              //构造函数
	{
		m_ErrorID = 1;
		strcpy(m_Error,"出现异常！ ");
	}
	int GetErrorID(){ return m_ErrorID; }         //获取异常 ID
	char * GetError(){ return m_Error; }          //获取异常信息
};
int main(int argc, char* argv[])
{
	try
	{
		throw (new CCustomError());               //抛出异常
	}
	catch(CCustomError* error)
	{
		//输出异常信息
		cout << "异常 ID：" << error->GetErrorID() << endl;
		cout << "异常信息：" << error->GetError() << endl;
	}
	return 0;
}
```

在上面的代码中可以看到 try 语句块中用于捕获 throw 所抛出的异常。对于 throw 异常的抛出，可以直接写在 try 语句块的内部，也可以写在函数或类方法的内部，但函数或方法必须写在 try 语句块的内部才可以捕获到异常。程序运行结果如图 14.5 所示。



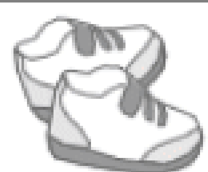
图 14.5　捕获一个异常

异常处理器可以成组地出现，同时根据 try 语句块获取的异常信息处理不同的异常。

【例 14.5】 获取不同异常的 try…catch。

实例位置：光盘\MR\Instance\14\14.5

```
int main(int argc, char* argv[])
{
	try
	{
		throw "字符串异常！ ";
		//throw (new CCustomError());          //抛出异常
	}
	catch(CCustomError* error)
	{
		//输出异常信息
		cout << "异常 ID：" << error->GetErrorID() << endl;
		cout << "异常信息：" << error->GetError() << endl;
	}
	catch(char * error)
```



```
    {
        cout << "异常信息：" << error << endl;
    }
    return 0;
}
```



Note

程序运行结果如图 14.6 和图 14.7 所示。



图 14.6　捕捉异常 1



图 14.7　捕捉异常 2

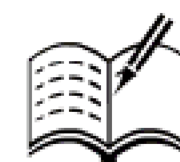
有时并不一定在列出的异常处理中包含所有可能发生的异常类型，所以 C++提供了可以处理任何类型异常的方法，就是在 catch 后面的括号内添加“…”。代码如下：

```
int main(int argc, char* argv[])
{
    try
    {
        throw "字符串异常！";
        //throw (new CCustomError());          //抛出异常
    }
    catch(CCustomError* error)
    {
        //输出异常信息
        cout << "异常 ID：" << error->GetErrorID() << endl;
        cout << "异常信息：" << error->GetError() << endl;
    }
    catch(char * error)
    {
        cout << "异常信息：" << error << endl;
    }
    catch(…)
    {
        cout << "未知异常信息！" << endl;
    }
```



```
    return 0;
}
```

有时需要重新抛出刚接收到的异常，尤其是在程序无法得到有关异常的信息而用省略号捕获任意的异常时。这些工作通过加入不带参数的 throw 即可完成：



```
catch (…) {
cout << "未知异常！ "<<endl;
throw ;
}
```

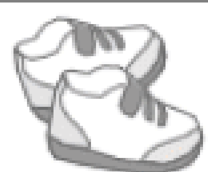
如果一个 catch 语句忽略了一个异常，那么这个异常将进入更高层的异常处理环境。由于每个异常抛出的对象是被保留的，所以更高层的异常处理器可抛出来自这个对象的所有信息。

14.5.3　异常匹配

当程序中有异常抛出时，异常处理系统会根据异常处理器的顺序找到最近的异常处理块，并不会搜索更多的异常处理块。

异常匹配并不要求异常与异常处理器进行完美匹配，一个对象或一个派生类对象的引用将与基类处理器进行匹配。若抛出的是类对象的指针，则指针会匹配相应的对象类型，但不会自动转换成其他对象的类型。例如：

```
#include "stdafx.h"
class CExcept1{};
class CExcept2
{
public:
    CExcept2(CExcept1& e){}
};
int main(int argc, char* argv[])
{
    try
    {
        throw CExcept1();                    //抛出异常
    }
    catch (CExcept2)                         //捕获异常 2
    {
        printf("进入 CExcept2 异常处理器！ \n");
    }
    catch(CExcept1)                          //捕获异常 1
    {
        printf("进入 CExcept1 异常处理器！ \n");
    }
    return 0;
}
```



从上面代码可以认为第一个异常处理器会使用构造函数进行转换，将 CExcept1 转换为 CExcept2 对象，但实际上系统在异常处理期间并不会执行这样的转换，而是在 CExcept1 处终止。

通过下面的代码演示基类处理器如何捕获派生类的异常。



Note

【例 14.6】 捕捉派生类异常。

实例位置：光盘\MR\Instance\14\14.6

```
#include "stdafx.h"
#include <iostream>
using namespace std;
class CExcept
{
public:
	virtual char *GetError(){ return "基类处理器"; }
};
class CDerive : public CExcept
{
public:
	char *GetError(){ return "派生类处理器"; }
};
int main(int argc, char* argv[])
{
	try                                     //抛出异常
	{
		throw CDerive();
	}
	catch(CExcept)                          //捕获异常
	{
	cout << "进入基类处理器\n";
	}
	catch(CDerive)                          //捕获异常
	{
		cout << "进入派生类处理器\n";
	}
	return 0;
}
```

程序运行结果如图 14.8 所示。

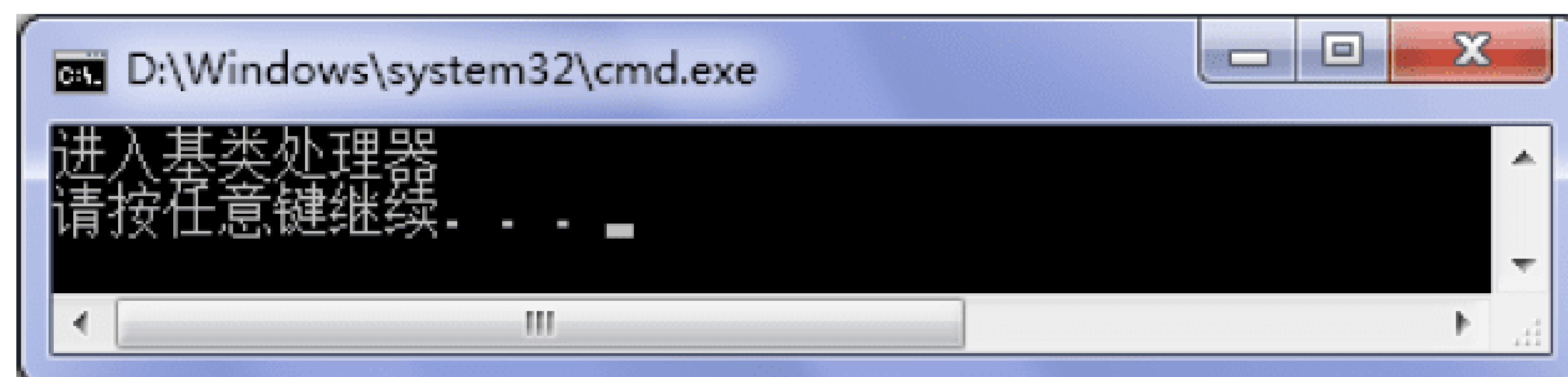
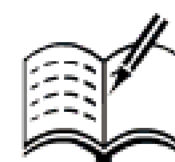


图 14.8　捕捉派生类异常

从上面的结果可以看出，虽然抛出的异常是 CDerive 类，但由于异常处理器的第一个是 CExcept 类，该类是 CDerive 类的基类，所以将进入此异常处理器内部。为了正确地进入指定的异常处理器，在对异常处理器进行排列时应将派生类排在前面，而将基类排在后面。



14.5.4　标准异常



用于 C++标准库的一些异常可以直接应用到程序中，应用标准异常类会比应用自定义异常类简单容易得多。如果系统提供的标准异常类不能满足需要，就不可以在这些标准异常类基础上进行派生。下面给出了 C++提供的一些标准异常：

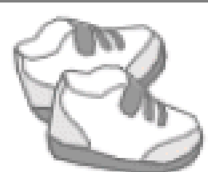
```
namespace std
{
     //exception 派生
     class logic_error;              //逻辑错误，在程序运行前可以检测出来
     //logic_error 派生
     class domain_error;             //违反了前置条件
     class invalid_argument;         //指出函数的一个无效参数
     class length_error;             //指出有一个超过类型 size_t 的最大可表现值长度的对象的企图
     class out_of_range;             //参数越界
     class bad_cast;                 //在运行时类型识别中有一个无效的 dynamic_cast 表达式
     class bad_typeid;               //报告在表达式 typeid(*p)中有一个空指针 p
     //exception 派生
     class runtime_error;            //运行时错误，仅在程序运行中检测到
     //runtime_error 派生
     class range_error;              //违反后置条件
     class overflow_error;           //报告一个算术溢出
     class bad_alloc;                //存储分配错误
}
```

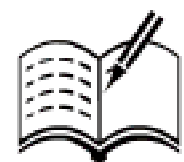
注意观察上述类的层次结构可以看出，标准异常都派生自一个公共的基类 exception。基类包含必要的多态性函数提供异常描述，可以被重载。下面是 exception 类的原型：

```
class exception
{
public:
    exception() throw();
    exception(const exception& rhs) throw();
    exception& operator=(const exception& rhs) throw();
    virtual ~exception() throw();
    virtual const char *what() const throw();
};
```

14.6　使用宏定义替换复杂的数据

在前面的学习中，经常遇到用#define 命令定义符号常量的情况，其实使用#define 命令就是要定义一个可替换的宏，宏定义是预处理命令的一种。它提供了一种可以替换源代码中字符串的





机制。根据宏定义中是否有参数，可以将其分为不带参数的宏定义和带参数的宏定义两种，下面分别进行介绍。

1．不带参数的宏定义

宏定义指令#define 用来定义一个标识符和一个字符串，以这个标识符来代表这个字符串，在程序中每次遇到该标识符时，就用所定义的字符串替换它。它的作用相当于给指定的字符串起一个别名。

不带参数的宏定义一般形式如下：

```
#define   宏名   字符串
```

#表示这是一条预处理命令。

宏名是一个标识符，必须符合 C 语言标识符的规定。

字符串可以是常数、表达式、格式字符串等。

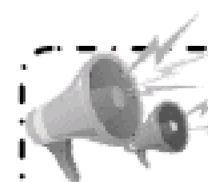
例如：

```
#define PI 3.14159
```

它的作用是在该程序中用 PI 替代 3.14159，在编译预处理时，每当在源程序中遇到 PI 就自动用 3.14159 代替。

使用#define 进行宏定义的好处是，需要改变一个常量时只需改变#define 命令行，整个程序的常量都会改变，大大提高了程序的灵活性。

宏名要简单且意义明确，一般习惯用大写字母表示，以便与变量名相区别。



注意：

宏定义不是 C 语句，不需要在行末加分号。

宏名定义后，即可成为其他宏名定义中的一部分。例如，下面代码定义了正方形的边长 SIDE、周长 PERIMETER 及面积 AREA 的值。

```
#define   SIDE   5
#define   PERIMETER   4*SIDE
#define   AREA   SIDE*SIDE
```

前面强调过宏替换是以字符串代替标识符。因此，如果希望定义一个标准的邀请语，可编写如下代码：

```
#define   STANDARD   "You are welcome to join us."
printf(STANDARD);
```

编译程序遇到标识符 STANDARD 时，就用“You are welcome to join us.”替换。

对于以上的编译程序，与 printf 语句的如下形式是等效的：

```
printf("You are welcome to join us.");
```





关于不带参数的宏定义有以下几点需要强调。

（1）如果在字符串中含有宏名，则不进行替换。例如：

```
#include "stdafx.h"
#define TEST "this is an example"
void main()
{
    char exp[30]="This TEST is not that TEST";     /*定义字符数组并赋初值*/
    printf("%s\n",exp);
}
```

该段代码的输入结果如图 14.9 所示。



图 14.9　在字符串中含有宏名

注意，上面程序字符串中的 TEST 并没有用“this is an example”来替换，所以说如果字符串中含有宏名，则不进行替换。

（2）如果字符串长于一行，可以在该行末尾用一反斜杠（\）续行。

（3）#define 命令出现在程序中函数的外面，宏名的有效范围为定义命令之后到此源文件结束。



注意：

在编写程序时通常将所有的#define 放到文件的开始处或独立的文件中，而不是将它们分散到整个程序中。

（4）可以用#undef 命令终止宏定义的作用域。

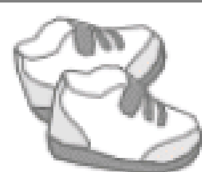
```
#include "stdafx.h"
#define TEST "this is an example"
main()
{
    printf(TEST);
    #undef TEST
}
```

（5）宏定义用于预处理命令，它不同于定义的变量，只作字符替换，不分配内存空间。

2．带参数的宏定义

带参数的宏定义不是简单的字符串替换，它还要进行参数替换。一般形式如下：

```
#define  宏名(参数表)字符串
```





Note

【例 14.7】 使用带参数的宏实现求两个数乘积。

实例位置：光盘\MR\Instance\14\14.7

```
#include "stdafx.h"
#define MUL(x,y) ((x)*(y))                          //定义两个数之和
int main()
{
	int a,b,c;
	printf("请输入两个整数：\n");
	scanf("%d%d",&a,&b);
	c=MUL(a,b);                                 //调用宏定义
	printf("两数乘积为：%d\n",c);
	return 0;
}
```

程序运行结果如图 14.10 所示。

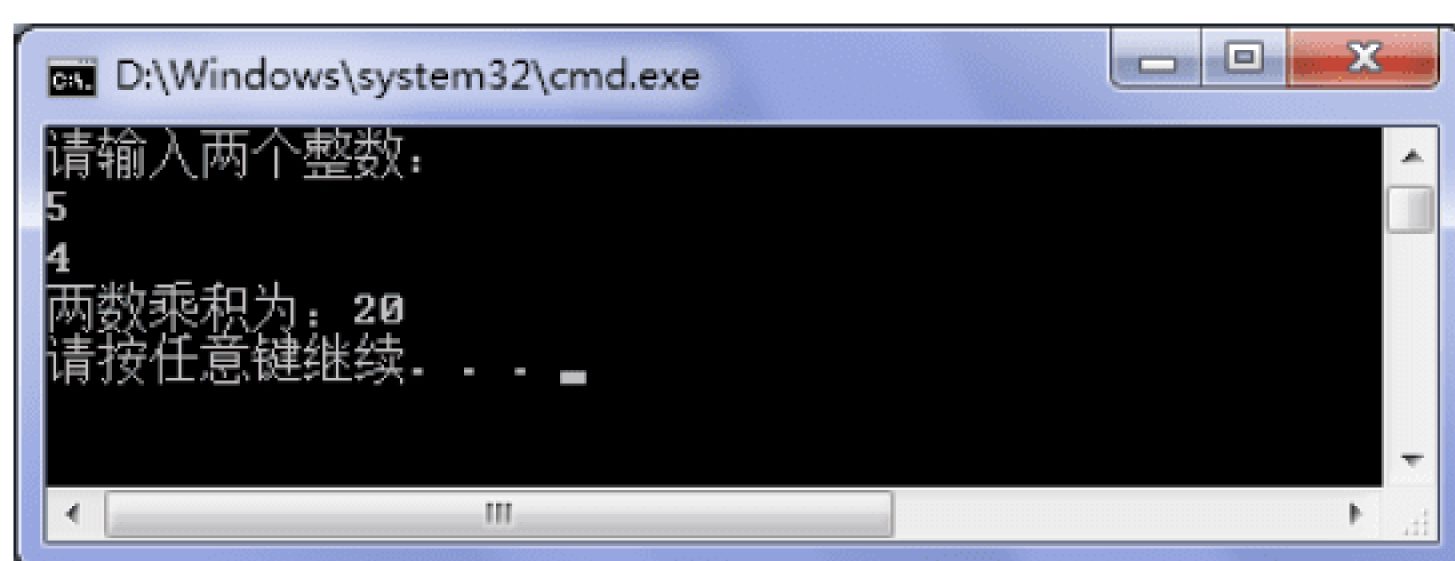


图 14.10　使用带参数的宏实现乘法运算

当编译该程序时，由 MUL(x,y)定义的表达式被替换，a 和 b 用作操作数，即“c=MUL(a,b);”语句被代替后变为如下形式：

```
c=((a)*(b));
```

用宏替换代替实在的函数的一个好处是，宏替换增加了代码的速度，因为不存在函数调用。但增加速度也有代价，即由于重复编码而增加了程序长度。

对于带参数的宏定义有以下几点需要强调。

（1）宏定义时参数要加括号。如不加括号，有时结果是正确的，有时结果便是错误的，那么什么时候是正确的，什么时候是错误的，下面具体进行说明。

如例 14.7 中，当参数 x=8，y=9 时，在参数不加括号的情况下调用 MUL(x,y)，可以正确地输出结果；当 x=8，y=5+4 时，若参数不加括号，即定义成“#define MUL(x,y) x*y”，这种情况下调用 MUL(x,y)，则输出的结果是错误的，因为此时调用的 MUL(x,y)执行情况如下：

```
c=8*5+4;
```

此时计算出的结果是 44，而实际上希望得出的结果是 72，为了避免出现上面这种情况，在进行宏定义时要在参数外面加上括号。

（2）宏扩展必须使用括号来保护表达式中低优先级的操作符，以确保调用时达到想要的效果。

例如，有如下宏定义：



```
#define SUB(x,y) (x)+(y)
```

则调用该宏定义时：

```
5*SUB（x,y）;
```



Note

会被扩展为：

```
5*(x)+(y);
```

而本意是希望得到：

```
5*((x)+(y));
```

解决的办法就是宏扩展时加上括号，就能避免这种错误发生。

（3）对带参数的宏的展开，只是将语句中的宏名后面括号内的实参字符串代替#define 命令行中的形参。

（4）在宏定义时，宏名与带参数的括号之间不可以加空格，否则将空格以后的字符都作为替代字符串的一部分。

（5）在带参宏定义中，形式参数不分配内存单元，因此不必做类型定义。

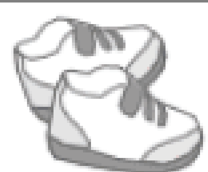
14.7　综合应用

14.7.1　扑克牌的牌面

【例 14.8】 假设扑克派 A～K 的牌面大小顺序为 3<4<5<…<Q<K<A<2。本实例设计一个扑克牌类，按照它们的牌面值可以比较大小。在扑克牌中 3 是最小的，可以将它作为枚举类型的第一个标识，其次是 4，最后是 2。在扑克牌内建立一个扑克牌牌面枚举类型的数据，并重载比较运算符达到比较的目的。关键代码如下：

实例位置：光盘\MR\Instance\14\14.8

```
//定义牌面枚举
enum Card_Value {card_3,card_4,card_5,card_6,card_7, card_8,
                    card_9,card_10,card_,Jcard_Q,card_K,card_A,card_2};
class Card
{
private:
    //牌面值，定义一个枚举变量 m_value
    Card_Value m_value;
public:
    Card(Card_Value c)
    {
        this->m_value =c;
    }
```





```
    //比较牌面值
    bool operator >(Card another)
    {
        if(Card.m_value>another.m_value)
        {
            return true;
        }
        return false;
    }
    bool operator <(Card another)
    {
        if(Card.m_value<another.m_value)
        {
            return true;
        }
        return false;
    }
    bool operator ==(Card another)
    {
        if(Card.m_value==another.m_value)
        {
            return true;
        }
        return false;
    }
}
```

程序运行结果如图 14.11 所示。

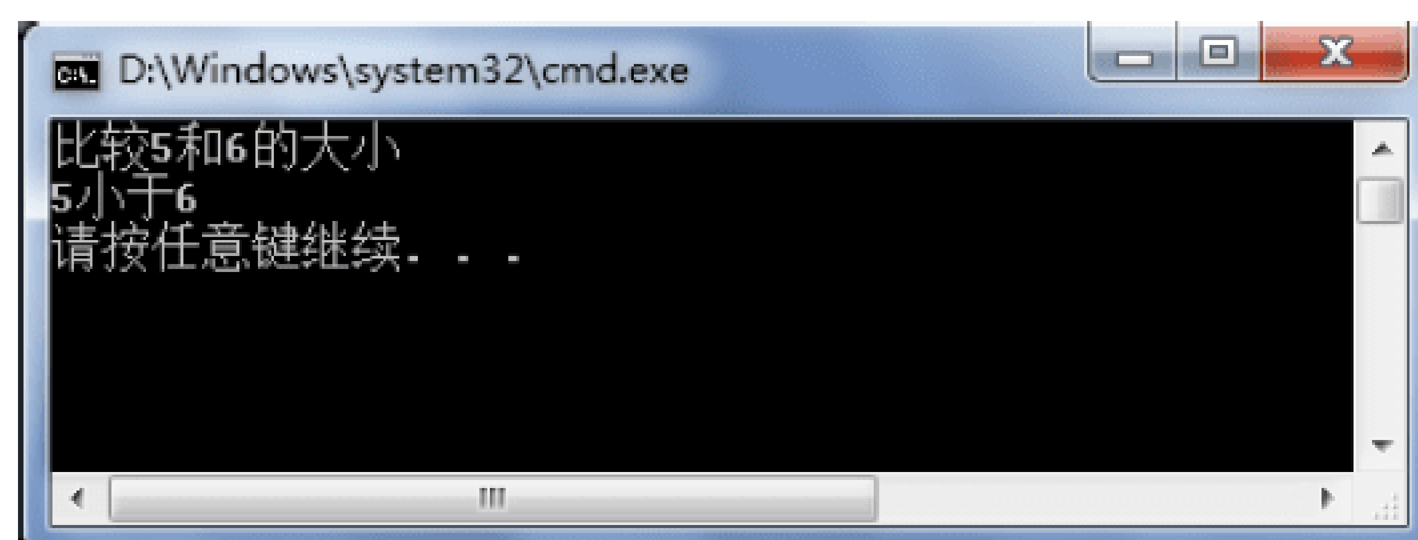


图 14.11　扑克牌的牌面

14.7.2　使用带参数的宏求圆面积

【例 14.9】 本实例将实现使用带参数的宏求圆面积。在程序内定义一个带参数的宏，使它能够计算圆的面积。关键代码如下：

实例位置：光盘\MR\Instance\14\14.9

```
//定义圆周率
#define PI 3.14
//定义带参数的宏求圆的面积
#define Area(r)    PI*(r)*(r)
```



程序运行结果如图 14.12 所示。

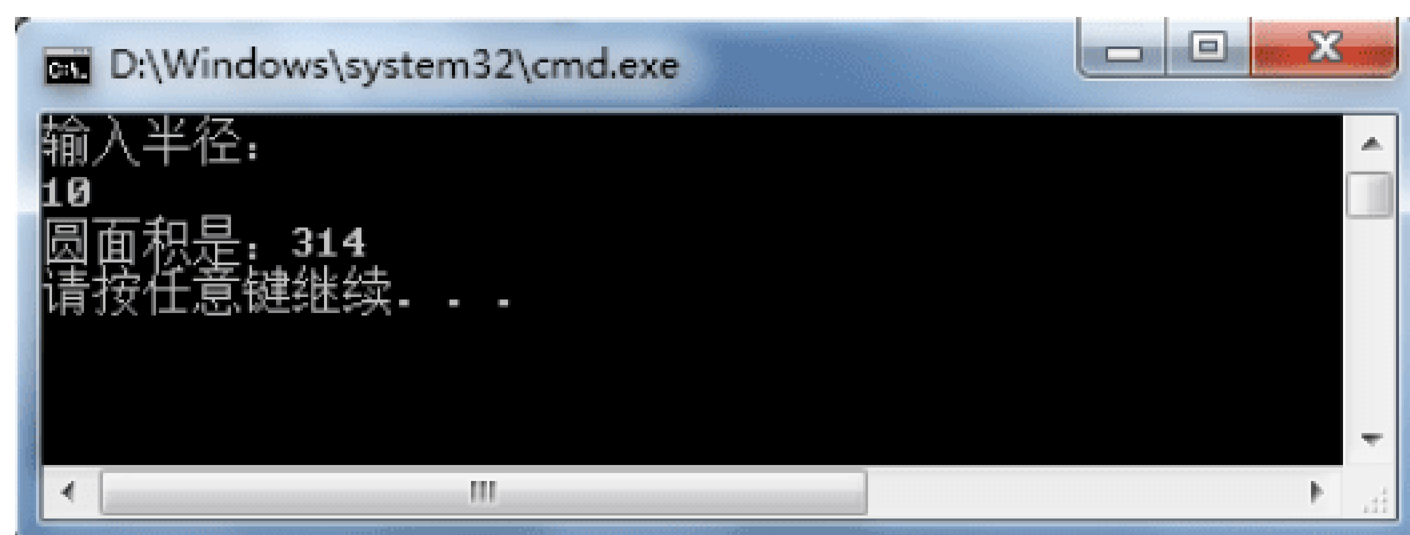


图 14.12　使用带参数的宏求圆面积



14.7.3　综合成绩

【例 14.10】 本例设计结构体，对学生进行综合评定。结构体包括学生期中成绩、期末成绩、平时考核成绩和综合成绩。输入学生各项成绩，按期中 30%、期末 50%、平时 20%计算学生的综合成绩。代码如下：

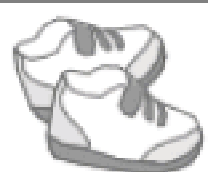
实例位置：光盘\MR\Instance\14\14.10

```
#include "stdafx.h"
#include <iostream>
using namespace std;
typedef struct student_score                    //定义结构体，用来存储期中、期末及平时考核
{
      int mid;                                  //期中
      int end;                                  //期末
      int ptime;                                //平时考核
      int sum;                                  //综合成绩
}score;
int _tmain(int argc, _TCHAR* argv[])
{
      score s;
      cout<<"输入期中、期末和平时考核成绩：\n";
      cin>>s.mid>>s.end>>s.ptime;
      s.sum = s.mid*0.3 + s.end*0.5 + s.ptime*0.2;
      cout<<"综合成绩："<<s.sum<<endl;          //综合成绩
      return 0;
}
```

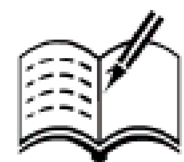
程序运行结果如图 14.13 所示。



图 14.13　综合成绩



14.8　本章常见错误



Note

14.8.1　注意带参数的宏

定义带参数的宏时，要注意加括号。宏是简单的替换和展开，它不是调用函数，不会自动将一个表达式当作一个整体。如下面的宏：

```
#define sub(a,b)   a-b              //宏 1
#define sub(a,b)   (a)-(b)          //宏 2
sub(10+1,6+1));                     //使用宏定义计算"10+1"减去"6+1"
```

我们本意是想计算 11 减去 7 的结果，但如果不加括号（如宏 1），会使计算结果出错。

宏 1 展开为 10+1−6+1，计算结果是 6；宏 2 展开为（10+1）－（6+1），结果是 4。

14.8.2　结构体成员的引用

如下结构体，对其成员的引用有以下 3 种方式：

```
struct                              //定义一个结构体类型
{
    int a;
    float b;
}item, *p;                          //定义一个结构体变量 item、结构体指针 p
p = &item;                          //让 p 指向结构体 item
```

以下 3 种方式都可以为成员 a 赋值：

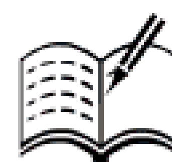
```
p->a = 10;
item.a = 10;
(*p).a = 10;
```

14.8.3　结构体字节对齐问题

结构体的内存布局依赖于 CPU、操作系统、编译器以及编译时的对齐选项。例如：

```
struct A
{
    char a;
    int b;
```






```
        double c;
        float d;
        int e;
}test;
```

此时结构体的大小应该是 sizeof(test) = 24。结构体中，最大的内存单元是 double，占 8 个字节，所以每次以 8 字节为单位分配空间，char 和 int 共占 5 个字节，分配一个 8 字节单元，double 独占一个 8 字节单元，float 和 int 共占 8 个字节，所以总共需要分配 3 个 8 字节单元，即 24 字节空间。

如果更改结构体中的各数据位置，整个结构体大小会变化。由此可见，合理地安排编译选项，可以有效节省内存。

14.8.4 用指针动态申请结构体内存时失败

例如下面的代码，为结构体申请内存时提示错误信息“error C2440: 'type cast' : cannot convert from 'void *' to 'struct x '”。

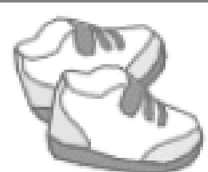
```
struct student
{
        int num;
};
int main()
{
        struct student *p = (struct student)malloc(sizeof( student));
}
```

首先使用 molloc 申请的内存是无数据类型的，使用时需要做类型转换，前面用 struct student* 类型的指针保存地址，应该将内存转换成 struct student*型。用 sizeof 测量结构体大小应该用 sizeof(struct student)。正确写法如下：

```
struct student *p = (struct student*)malloc(sizeof(struct student));
```

14.9 本 章 小 结

本章介绍了 C++中的结构体，讲述了类型别名和枚举类型的使用方法。在 C++中，类型推导是一个方便而实用的特性，它解决了对象创建时类名空间名称复杂所带来的麻烦。在程序中使用宏定义数据的值可以节省内存空间，还可以使改动代码变得更简单。程序中出现异常是不可避免的，异常处理则能够帮助程序开发人员尽快发现错误所在。为了减少错误的发生，应尽量掌握更多的异常处理方式。



14.10 跟我上机



Note

参考答案：光盘\MR\跟我上机

有时调用函数需要传递很多个参数，很不方便，此时可以使用结构体封装这些参数，实现一次传递多个参数。本例将设计一个结构体，封装 3 个不同类型的参数，调用函数 display，传递一个结构体变量 t，输出，F5 是调试运行。实现如下：

```
#include <iostream>
#include <string>
using std::cout;
using std::string;
using std::endl;
typedef struct                          //定义一个结构体类型，并用 typedef 重命名为 T
{
	int n;
	string s;
	char c;
}T;
void display(T &t)                      //输出显示函数
{
	cout<<t.c<<t.n<<t.s<<endl;
}
int main()
{
	T t;                            //定义结构体变量 t
	t.c = 'F';                      //结构体成员赋值
	t.n = 5;
	t.s = "是调试运行";
	display(t);                     //调用显示函数
	return 0;
}
```

第 15 章

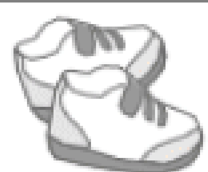
掌握 C++标准模板库

（视频讲解：29 分钟）

标准模板库 STL 的英文全称为 Standard Template Library，主要目的是为标准化组件提供类模板进行范型编程。STL 技术是对原有 C++技术的一种补充，具有通用性好、效率高、数据结构简单、安全机制完善等特点。STL 是一些容器的集合，这些容器在算法库的支持下使程序开发变得更加简单和高效。

本章能够完成的主要范例（已掌握的在方框中打勾）

- ☐ 容器的使用
- ☐ 使用 list 和 vector 中的迭代器
- ☐ 关联容器
- ☐ 使用 for_each 算法输出容器内的元素
- ☐ 应用 fill 算法对容器元素赋值
- ☐ 应用 sort 算法对迭代器所指明的范围内的元素排序
- ☐ 使用 transform 将指定容器范围中的元素执行指定操作
- ☐ 迭代输出信息



15.1　几种常见数据结构



15.1.1　简述 STL

标准模板库，即 STL（Standard Template Library），是根据本地 C++标准规范定义的一套功能强大、适用范围广的一套函数模板、类模板的库。它主要包含容器、算法、函数对象等内容。

☑　容器实现了多个类型的数据结构和一些相关操作。

☑　算法提供了排序、查找、替换功能等的函数。

☑　函数对象是 C++运算符重载的进一步应用，类模板实例化后很多工作都需要它的支持。

首先简单介绍一些数据结构的基本概念。如果读者学习过数据结构，请跳过本节。

15.1.2　顺序线性结构

数据结构本身是一种集合，包含的各项称为元素。

数组是顺序线性结构的实现，如图 15.1 所示，通过指针来访问所有元素（对象），之后对它执行相应的操作。



图 15.1　数组的结构示意图

15.1.3　基本操作

操作主要分为以下 4 种。

☑　查询：通过输入或者已知条件在数据结构中找到相应的元素。

☑　插入：在各个元素之间，逻辑插入一个元素。

☑　删除：在各个元素中，逻辑删除一个元素。

☑　修改：通过输入或者已知条件在数据结构中修改相应的元素。



注意：

逻辑插入意味着元素所在的储存单元并不一定真的发生插入操作，例如，数组本身所在的内存分布是连续的，无法在中间插入任何内存单元。实现插入的方法为：将数组所有元素和待插入的元素依照相应的次序复制给另外一个容量+1 的数组，之后销毁原来的内存空间。新的数组可以被认为是插入完成之后的数组（一般需要动态分配）。逻辑删除与它的原理相同。





15.1.4　栈结构

栈的结构如图 15.2 所示。它很像一个从上方打开的箱子，所有的对象都遵循着后进先出的原则。在栈中随机位置插入和删除的实现通常需要辅助方法实现，相对较为缓慢。若所储存的对象超过自身容量，则会出现栈溢出。

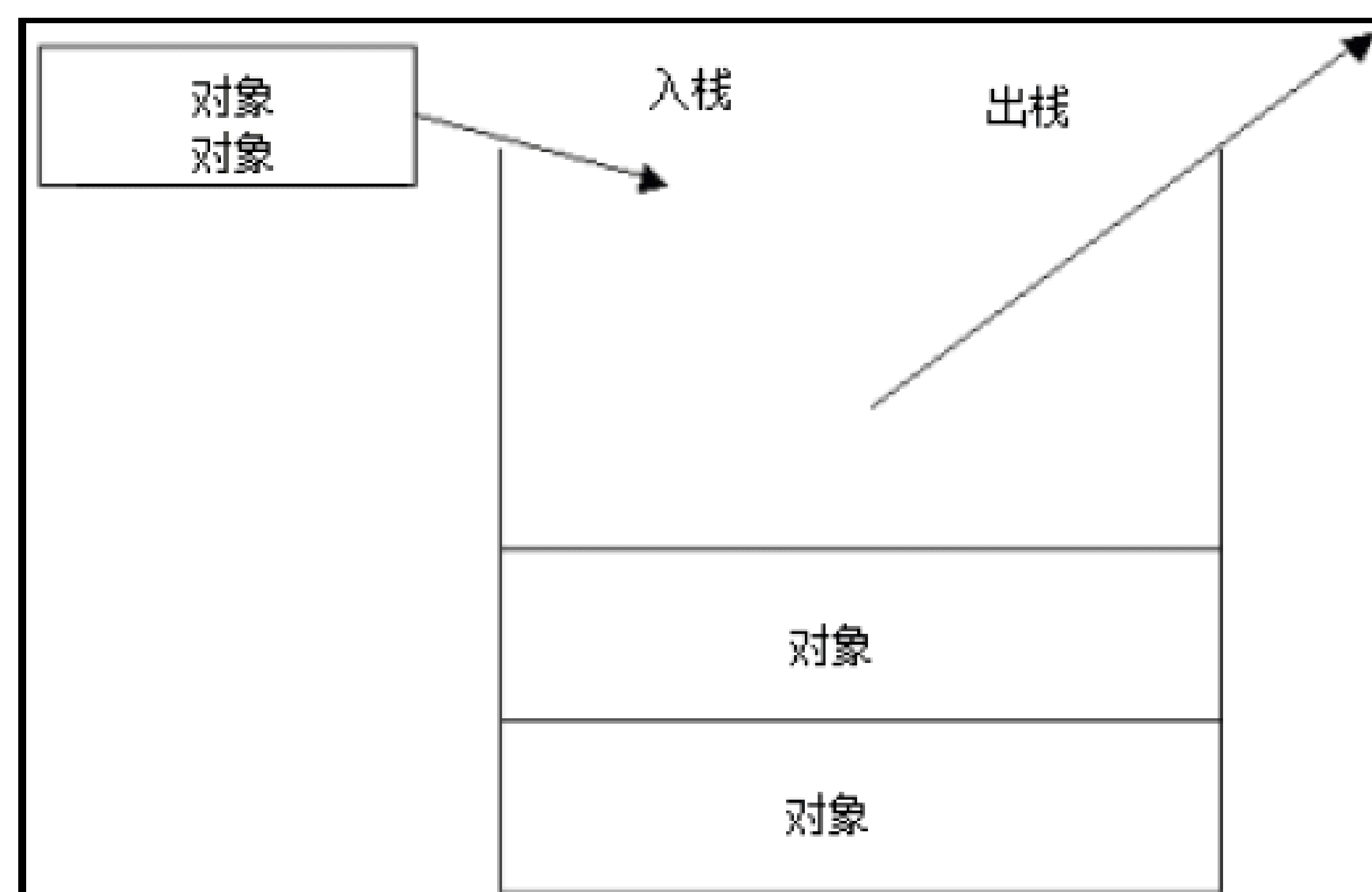


图 15.2　栈结构的示意图

15.1.5　队列结构

队列的结构和现实生活中的排队一样。所有对象只能从队列的后方加入，从前方离开队列，即先进先出。和栈相似的是，队列不擅长处理随机位置的插入和删除操作，如图 15.3 所示。

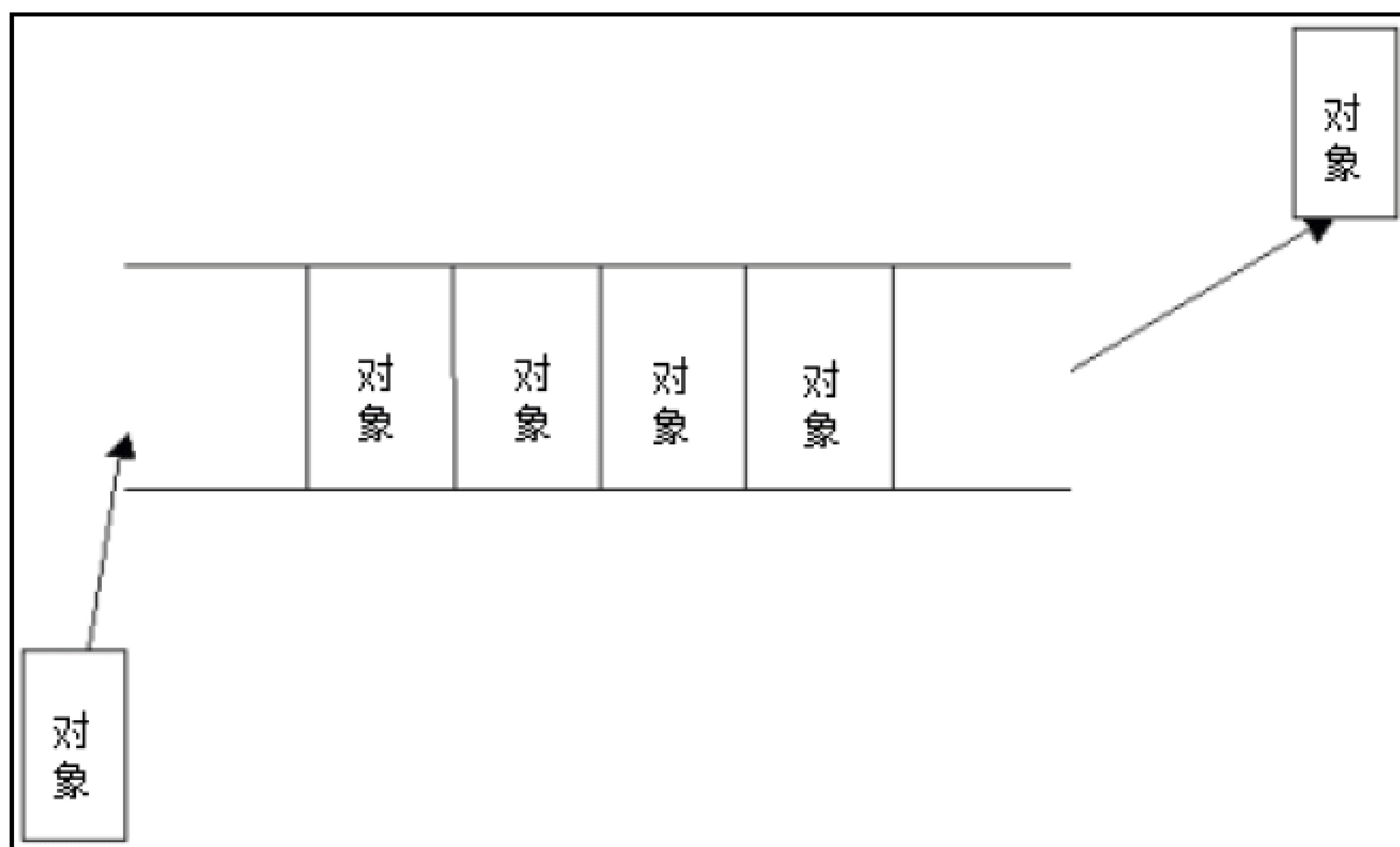
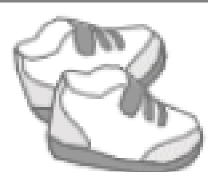


图 15.3　队列结构示意图

15.1.6　链表结构

链表与以上几种结构都是线性的结构。与它们不同的是，链表的储存方式不是顺序的。每个





Note

对象中含有下一个对象所在位置的信息（可认为是地址），链表的第一个元素的位置由头指针（也称为 Head）储存，链表尾部的元素储存的位置信息为空。由于链表的储存区不是连续的，所以无法通过指针偏移量的方案查找相应对象，只能够通过以遍历的方法访问：链表由头指针所储存的地址信息访问到第一个元素，之后依次由所储存的位置信息（地址）访问后面的元素。链表的优点是在随机位置的插入和删除较为迅速，如图 15.4 所示。

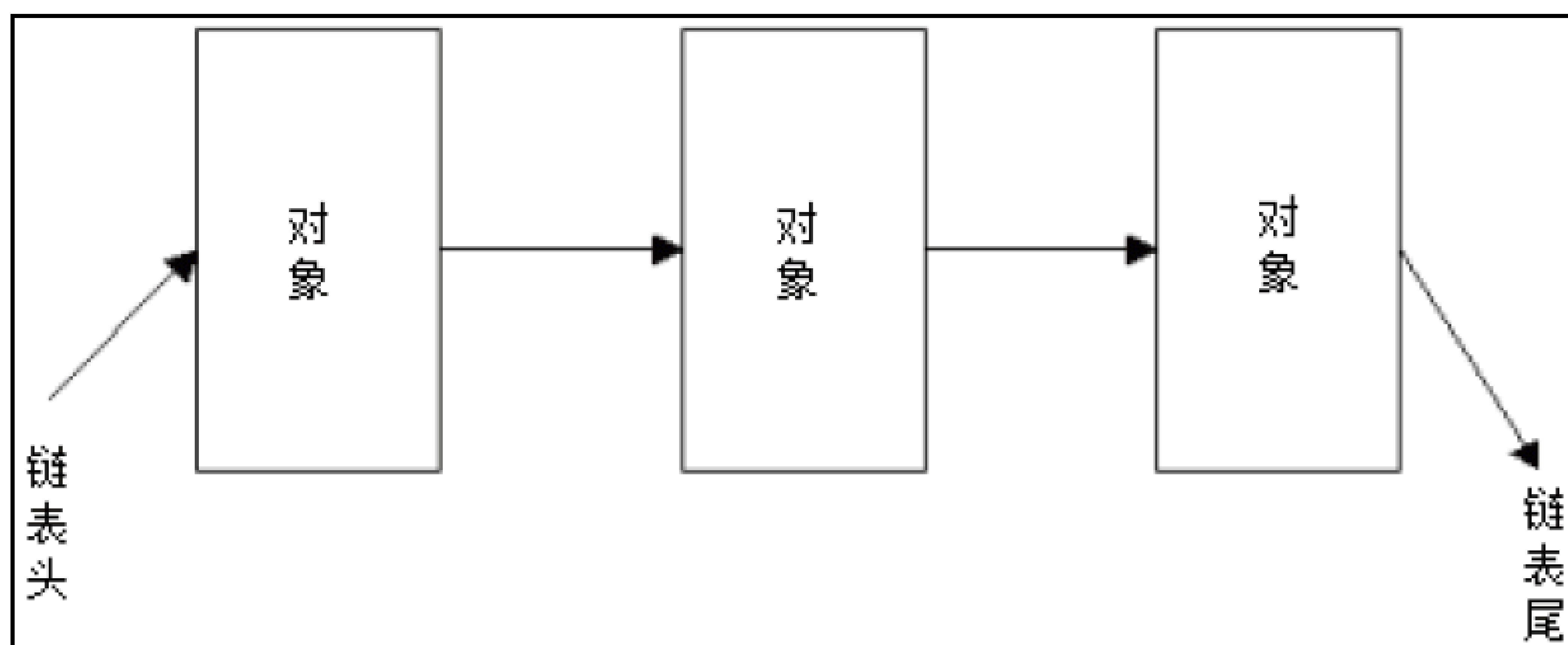


图 15.4　链表结构示意图

链表可分为单向链表、双向链表和循环链表。

15.1.7　图结构

图结构是一种复杂的数据结构，每个对象都可以和其他对象相关联。图一般采取映射的方式表示内部元素。映射通常指的是事物间的联系，如图 15.5 所示，每条边代表着两个物体所对应的关系。使用表达式<k,t>表示的是这种关系，k 和 t 可以代表线的两端，也可以是线与物体，随着不同的定义，所代表的关系也不同。以上简单介绍了几种常用的数据结构，在标准模板库中，容器类对它们进行了实现、扩展和封装。

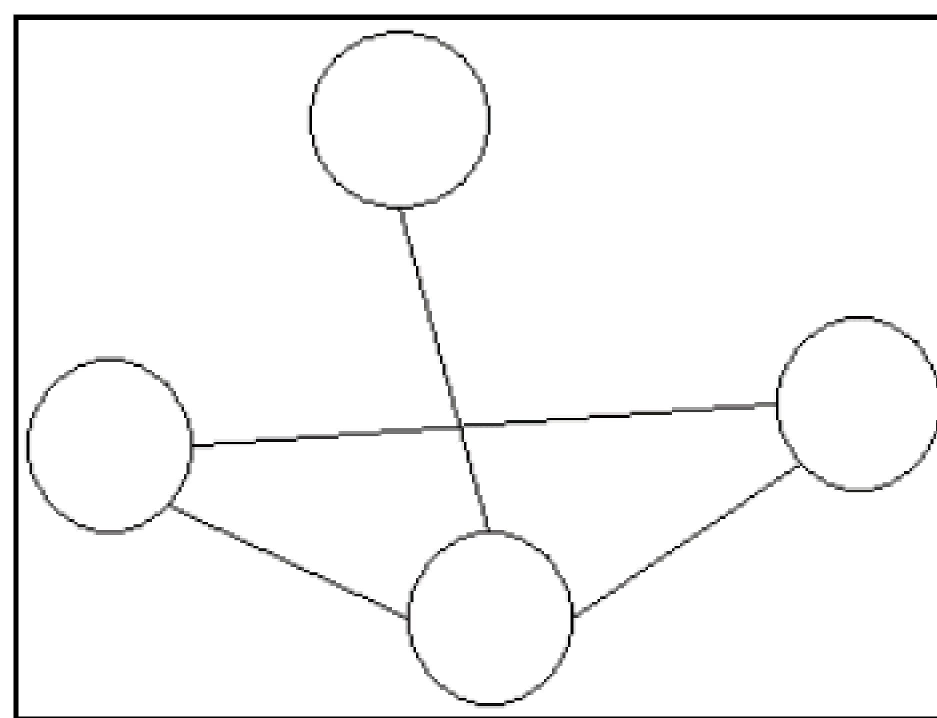


图 15.5　图结构示例

15.2　使用容器管理数据

15.2.1　对比容器适配器与容器

标准模板库的容器适配器与容器都是用来储存和组织对象的模板类。容器适配器与容器相比，限制的条件更多。容器适配器定义在相应的头文件中，如表 15.1 所示。



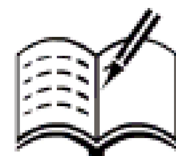


表 15.1　容器适配器头文件内容

头　文　件	内　　容
queue	定义了一些具有队列结构特征的类模板。其中包含了 queue<T>，是一种单向队列。priority_queue 排列自身对象，最大的值会被放在队列前端
stack	包含了 stack<T>类模板，具有栈数据结构的特征

容器适配器还被定义在容器的头文件中，这通常与容器的内部实现有关，如表 15.2 所示。

表 15.2　容器头文件内容

头　文　件	内　　容
vector	vector<T>是一个在必要时能够自动增大容量的数组，在随机位置上插入元素会花费很大的系统开销。其中定义了对应的适配器 queue
dqueue	dqueue<T>是一个双向队列，与 vector 作用相似。但多出了从队列前加入元素的特性。其中也定义了对应的适配器 queue 和 stack
list	list<T>是一种双向的链表。定义了适配器 stack
map	map<K,T>是一种关联容器。K 表示关联的对象 T 所在 map 中位置的信息，值必须唯一
set	set<T>表示的是一一对应的关系。T 就是这种关系的象征，它在 set 中唯一并且不能够被直接修改。只能够删除，之后加入新的对象来达到目的

在 C++中使用标准模板库提供的容器，需要再加入相应的头文件并使用名称空间 std。容器适配器与容器在使用限制上的最大区别在于是否支持迭代器。迭代器的行为类似于指针，通过它能够遍历容器中的所有元素，但容器适配器不支持它。通常情况下，我们更倾向于使用容器而非容器适配器。

15.2.2　对比迭代器与容器

标准模板库中提供了 4 种迭代器，如表 15.3 所示。

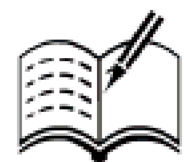
表 15.3　迭代器的分类

迭　代　器	功　　能
输入和输出迭代器	支持对象序列的读/写，仅能使用一次（不可重用）。支持了自加运算符++来获得一个新的迭代，这样它才可以进行下一次读/写
前向迭代器	支持输入和输出迭代器的功能，还可进行对象的访问和储存操作。前向迭代器可以重用，用来遍历容器
双向迭代器	双向迭代器包含了前向迭代器的功能。支持自减运算符--，使它能够反向遍历容器
随机迭代器	包含了以上所有迭代器的功能。重载了加、减运算符，可以对容器内任何元素进行随机访问。它还支持索引运算符和比较运算符

这 4 种迭代器在功能上都是“向上兼容”的，越来越强大。容器自身的迭代器种类是依照容器的结构来决定的，vector 包含的迭代器是随机迭代器类，list 包含的是双向迭代器，在 queue 中的迭代器则是前向迭代器。



15.2.3 vector 容器



Note

向量（vector）是一种随机访问的数组类型，提供了对数组元素的快速、随机访问，以及在序列尾部快速、随机地插入和删除操作。它是大小可变的向量，在需要时可以改变其大小。

使用向量类模板需要创建 vector 对象，创建 vector 对象有以下几种方法：

```
std::vector<type> name;
```

该方法创建了一个名为 name 的空 vector 对象，该对象可容纳类型为 type 的数据。例如，为整型值创建一个空 std::vector 对象可以使用这样的语句：

```
std::vector<int> intvector;
std::vector<type> name(size);
```

该方法用来初始化具有 size 元素个数的 vector 对象：

```
std::vector<type> name(size,value);
```

该方法用来初始化具有 size 元素个数的 vector 对象，并将对象的初始值设为 value：

```
std::vector<type> name(myvector);
```

该方法使用复制构造函数，用现有的向量 myvector 创建了一个 vector 对象：

```
std::vector<type> name(first,last);
```

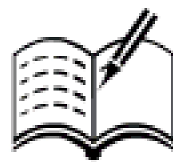
该方法创建了元素在指定范围内的向量，first 代表起始范围，last 代表结束范围。

vector 对象的主要成员继承于随机接入容器和反向插入序列，主要成员函数及说明如表 15.4 所示。

表 15.4 vector 对象主要成员函数及说明

函　数	说　明
assign(first,last)	用迭代器 first 和 last 所辖范围内的元素替换向量元素
assign(num.val)	用 val 的 num 个副本替换向量元素
at(n)	返回向量中第 n 个位置元素的值
back	返回对向量末尾元素的引用
begin	返回指向向量中第一个元素的迭代器
capcity	返回当前向量最多可以容纳的元素个数
clear	删除向量中所有元素
empty	如果向量为空，则返回 true 值
end	返回指向向量中最后一个元素的迭代器
erase(start,end)	删除迭代器 start 和 end 所辖范围内的向量元素
erase(i)	删除迭代器 i 所指向的向量元素
front	返回对向量起始元素的引用





续表

函　数	说　明
insert(i,x)	把值 x 插入向量中由迭代器 i 所指明的位置
insert(i,start,end)	把迭代器 start 和 end 所辖范围内的元素插入到向量中由迭代器 i 所指明的位置
insert(i,n,x)	把 x 的 n 个副本插入到向量中由迭代器 i 所指明的位置
max_size	返回向量的最大容量（最多可以容纳的元素个数）
pop_back	删除向量最后一个元素
push_back(x)	把值 x 放在向量末尾
rbegin	返回一个反向迭代器，指向向量末尾元素之后
rend	返回一个反向迭代器，指向向量起始元素
reverse	颠倒元素的顺序
resize(n,x)	重新设置向量大小 n，新元素的值初始化为 x
size	返回向量的大小（元素的个数）
swap(vector)	交换两个向量的内容

下面用一个实例演示 vector 类的使用方法。

【例 15.1】 vector 的操作方法。

实例位置：光盘\MR\Instance\15\15.1

```
#include "stdafx.h"
#include <iostream>
#include <vector>
using std::cout;
using std::endl;
using std::vector;
int main(int argc, _TCHAR* argv[])
{
	vector<int> v1,v2;                      //定义两个容器
	v1.reserve(10);                         //手动分配空间，设置容器元素最小值
	v2.reserve(10);
	v1 = vector<int>(8,7);
	int array[8]= {1,2,3,4,5,6,7,8};        //定义数组
	v2 = vector<int>(array,array+8);;       //给 v2 赋值
	cout<<"v1 容量"<<v1.capacity()<<endl;
	cout<<"v1 当前各项:"<<endl;
	for(decltype(v2.size()) i = 0;i<v1.size();i++)
	{
		cout<<" "<<v1[i];
	}
	cout<<endl;
	cout<<"v2 容量"<<v2.capacity()<<endl;
	cout<<"v2 当前各项:"<<endl;
	for(vector<int>::size_type i = 0;i<v1.size();i++)
	{
		cout<<" "<<v2[i];
	}
```


Note

```
cout<<endl;
v1.resize(0);
cout<<"v1 的容量通过 resize 函数变成 0"<<endl;
if(!v1.empty())
	cout<<"v1 容量"<<v1.capacity()<<endl;
else
	cout<<"v1 是空的"<<endl;
cout<<"将 v1 容量扩展为 8"<<endl;
v1.resize(8);
cout<<"v1 当前各项:"<<endl;
for(decltype(v1.size()) i = 0;i<v1.size();i++)
{
	cout<<" "<<v1[i];
}
cout<<endl;
v1.swap(v2);
cout<<"v1 与 v2 swap 了"<<endl;
cout<<"v1 当前各项:"<<endl;
cout<<"v1 容量"<<v1.capacity()<<endl;
for(decltype(v1.size()) i = 0;i<v1.size();i++)
{
	cout<<" "<<v1[i];
}
cout<<endl;
v1.push_back(3);
cout<<"从 v1 后边加入了元素 3"<<endl;
cout<<"v1 容量"<<v1.capacity()<<endl;
for(decltype(v1.size()) i = 0;i<v1.size();i++)
{
	cout<<" "<<v1[i];
}
cout<<endl;
v1.erase(v1.end()-2);
cout<<"删除了倒数第二个元素"<<endl;
cout<<"v1 容量"<<v1.capacity()<<endl;
cout<<"v1 当前各项:"<<endl;
for(vector<int>::size_type i = 0;i<v1.size();i++)
{
	cout<<" "<<v1[i];
}
cout<<endl;
v1.pop_back();
	cout<<"v1 通过栈操作 pop_back 放走了最后的元素"<<endl;
cout<<"v1 当前各项:"<<endl;
cout<<"v1 容量"<<v1.capacity()<<endl;
for(vector<int>::size_type i = 0;i<v1.size();i++)
{
	cout<<" "<<v1[i];
}
cout<<endl;
```





Note

```
	return 0;
}
```

程序运行结果如图 15.6 所示。

```
D:\Windows\system32\cmd.exe
 7 7 7 7 7 7 7 7
v2容量8
v2当前各项:
 1 2 3 4 5 6 7 8
v1的容量通过resize函数变成0
v1是空的
将v1容量扩展为8
v1当前各项:
 0 0 0 0 0 0 0 0
v1与v2 swap了
v1当前各项:
v1容量8
 1 2 3 4 5 6 7 8
从v1后边加入了元素3
v1容量12
 1 2 3 4 5 6 7 8 3
删除了倒数第二个元素
v1容量12
v1当前各项:
 1 2 3 4 5 6 7 3
v1通过栈操作pop_back放走了最后的元素
v1当前各项:
v1容量12
 1 2 3 4 5 6 7
请按任意键继续. . .
```

图 15.6　vector 的操作方法

实例演示了 vector<int>容器的初始化，以及插入、删除等操作。在本例中 v1 和 v2 均用 resize 分配了空间。当分配的空间小于自身原来的空间大小时，删除原来的末尾元素。当分配的空间大于自身的空间时自动在末尾元素后边添加相应个数的 0 值。同理，若 vector 模板使用的是某一个类，则增加的会是以默认构造函数创建的对象。同时可以看到，向 v1 添加元素时，v1 的容量从 8 增加到了 12，这个就是在 vector 提供的特性，在需要时可以扩大自身的容量。



注意：

虽然 vector 支持 insert 函数插入，但与链表数据结构的容器比较而言效率较差，不推荐经常使用。

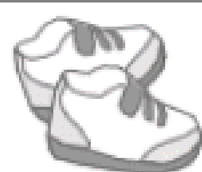
15.2.4　list 容器

链表（list）即双向链表容器，它不支持随机访问，访问链表元素需要指针从链表的某个端点开始，插入和删除操作所花费的时间是固定的，和该元素在链表中的位置无关。list 在任何位置插入和删除动作都很快，不像 vector 只在末尾进行操作。

使用链表类模板需要创建 list 对象，创建 list 对象有以下几种方法：

```
std::list<type> name;
```

该方法创建了一个名为 name 的空:list 对象，该对象可容纳数据类型为 type 的数据。例如，为整型值创建一个空 std::vector 对象可以使用这样的语句：std::list<type> name(size);。





Note

该方法初始化具有 size 元素个数的 list 对象：

```
std::list<type> name(size,value);
```

该方法初始化具有 size 元素个数的 list 对象，并将对象的每个元素设为 value：

```
std::list<type> name(mylist);
```

该方法使用复制构造函数，用现有的链表 mylist 创建了一个 list 对象：

```
std::list<type> name(first,last);
```

该方法创建了元素在指定范围内的链表，first 代表起始范围，last 代表结束范围。

list 对象的主要成员函数及说明如表 15.5 所示。

表 15.5　list 对象的主要成员函数及说明

函　　数	说　　明
assign(first,last)	用迭代器 first 和 last 所辖范围内的元素替换链表元素
assign(num,val)	用 val 的 num 个副本替换链表元素
back	返回一个对链表最后一个元素的引用
begin	返回指向链表中第一个元素的迭代器
clear	删除双链表中的所有元素
empty	如果链表为空，则返回 true 值
end	返回指向链表最后一个元素的迭代器
erase(start,end)	删除迭代器 start 和 end 所辖范围内的链表元素
erase(i)	删除迭代器 i 所指向的链表元素
front	返回一个对链表第一个元素的引用
insert(i,x)	把值 x 插入链表中由迭代器 i 所指明的位置
insert(i,start,end)	把迭代器 start 和 end 所辖范围内的元素插入到链表中由迭代器 i 所指明的位置
insert(i,n,x)	把 x 的 n 个副本插入到链表中由迭代器 i 所指明的位置
max_size	返回链表的最大容量（最多可以容纳的元素个数）
pop_back	删除链表最后一个元素
pop_front	删除链表第一个元素
push_back(x)	把值 x 放在链表末尾
push_front(x)	把值 x 放在链表开始
rbegin	返回一个反向迭代器，指向链表最后一个元素之后
rend	返回一个反向迭代器，指向链表第一个元素
resize(n,x)	重新设置链表大小 n，新元素的值初始化为 x
reverse	颠倒链表元素的顺序
size	返回链表的大小（元素的个数）
swap(listref)	交换两个链表的内容
swap(vector)	交换两个链表的内容

可以发现，list<T>所支持的操作与 vector<T>很相近，但这些操作的实现原理不尽相同，执



行效率也不一样。list（双向链表）的优点是插入元素的效率很高，缺点是不支持随机访问。也就是说，链表无法像数组一样通过索引来访问。例如：

```
list<int>    list1 (first,last);                    //初始化
list[i] = 3;                                        //错误！！无法使用数组符号[]
```



Note

对 list 各个元素的访问，通常使用的是迭代器。

迭代器的使用方法类似于指针，下面用一个实例演示用迭代器访问 list 中的元素。

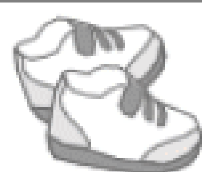
【例 15.2】 list 和 vector 中的迭代器。

实例位置：光盘\MR\Instance\15\15.2

```
#include "stdafx.h"
#include <iostream>
#include <list>
#include <vector>
using std::vector;
using std::list;
using std::cout;
using std::endl;
int main()
{
	cout<<"使用未排序储存 0-9 的数组初始化 list1"<<endl;
	int array[10] = {1,3,5,7,8,9,2,4,6,0};
	list<int> list1(array,array+10);
	cout<<"list1 调用 sort 方法排序"<<endl;
	list1.sort();
	list<int>::iterator iter = list1.begin();
	//iter =iter+5      list 的 iter 不支持+运算符
	cout<<"通过迭代器访问 list 双向链表中从头开始向后的第 4 个元素"<<endl;
	for(int i = 0;i<3;i++)
	{
		iter++;
	}
	cout<<*iter<<endl;
	list1.insert(list1.end(),13);
	cout<<"在末尾插入数字 13"<<endl;
	for(auto it = list1.begin();it != list1.end();it++)
	{
		cout<<" "<<*it;
	}
}
```

程序运行结果如图 15.7 所示。

通过程序可以观察到，迭代器 iterator 类和指针用法很相似，支持自增操作符，并且通过“*”可以访问相应的对象内容。但 list 中的迭代器不支持“+”号运算符，而指针与 vector 中的迭代器都支持。





Note

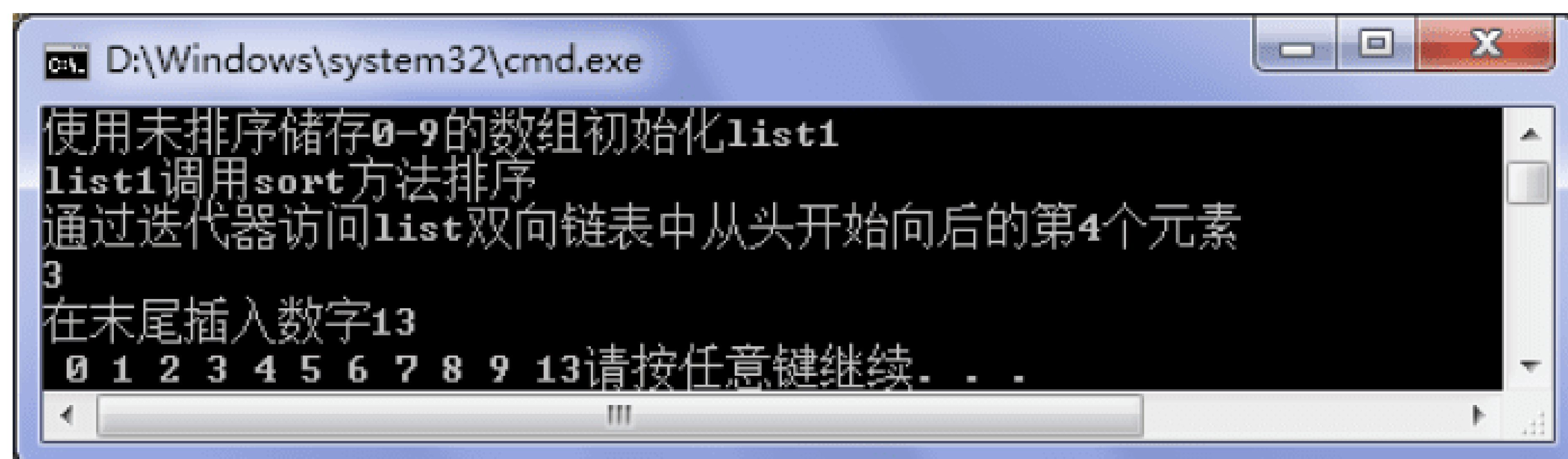


图 15.7　迭代器的应用

15.2.5　关联容器

关联容器也称为结合容器，是图结构的实现。它的每个对象都代表一组关系映射<K,T>。其中 K 代表 key，即键值。通过键值可以迅速地找到它的关联条目 T。前边所介绍的 list 和 vector 容器都是序列容器，它们经常表示的是有序的对象列表，而关联容器则代表着对象关系的集合。下面介绍两种关联容器 pair 和 map。

pair<K,T>模板表示的是 K 类对象与 T 类对象的关联。举例来说，pair<int,string>可以代表的是数字类 ID 和员工名字的一组关系，通过 ID 即可查找到员工对象。

pair 容器可以通过以下方式初始化：

```
//t1、t2 分别是 T1、T2 类的对象
pair<T1,T2> pair1(t1,t2);
```

在 pair 模板中包含两个 public 成员变量 first 和 second。first 是 pair 中的键，second 是 pair 中的关联条目。

map<K,T>模板包含的对象全部是与模板对应类型的 pair<K,T>模板对象，即 map 是包含映射关系 pair 的容器。map 对象主要成员函数及说明如表 15.6 所示。

表 15.6　map 对象主要成员函数及说明

函　　数	说　　明
begin	返回指向集合中第一个元素的迭代器
clear	删除集合中所有元素
empty	如果集合为空，则返回 true 值
end	返回指向集合中最后一个元素的迭代器
equal_range(x)	返回表示 x 下界和上界的两个迭代器，下界表示集合中第一个值等于 x 的元素，上界表示第一个值大于 x 的元素
erase(i)	删除由迭代器 i 所指向的集合元素
erase(start,end)	删除由迭代器 start 和 end 所指范围内的集合元素
erase(x)	删除集合中值为 x 的元素
find(x)	返回一个指向的迭代器。如果 x 不存在，返回的迭代器等于 end
lower_bound(x)	返回一个迭代器，指向位于 x 之前且紧邻 x 的元素
max_size	返回集合的最大容量
rbegin	返回一个反向迭代器，指向集合最后一个元素



续表

函　　数	说　　明
rend	返回一个反向迭代器，指向集合的第一个元素
size	返回集合的大小
swap()	交换两个集合的内容
upper_bound()	返回一个指向 x 的迭代器
value_comp	返回 value_compare 类型的对象，该对象用于判断集合中元素的先后次序



通过 insert 方法可以在 map 容器中插入一个 pair 对象，通过 erase 删除一个 pair 对象。

【例 15.3】 关联容器的操作。

实例位置：光盘\MR\Instance\15\15.3

```
#include "stdafx.h"
#include <map>
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
int main()
{
	int i = 1;
	string name("jack");
	pair<int,string> pair1(1,name);
	pair<int,string> pair2(2,"老张");
	map<int,string> map1;
	map1.insert(pair1);
	map1.insert(pair2);
	cout<<"通过 find 函数返回的迭代器访问键值为 1 的关联条目"<<endl;
	auto it = map1.find(1);
	if(it!=map1.end())
	{
		string name = it->second;
		cout<<name<<endl;
	}
	cout<<"访问键值为 2 的关联条目"<<endl;
	cout<<"名字:"<<map1[2]<<endl;
	cout<<"删除键值为 2 的 pair"<<endl;
	map1.erase(2);
	cout<<"访问键值为 2 的关联条目"<<endl;
	cout<<"名字:"<<map1[2]<<endl;
	return 0;
}
```

程序运行结果如图 15.8 所示。





Note

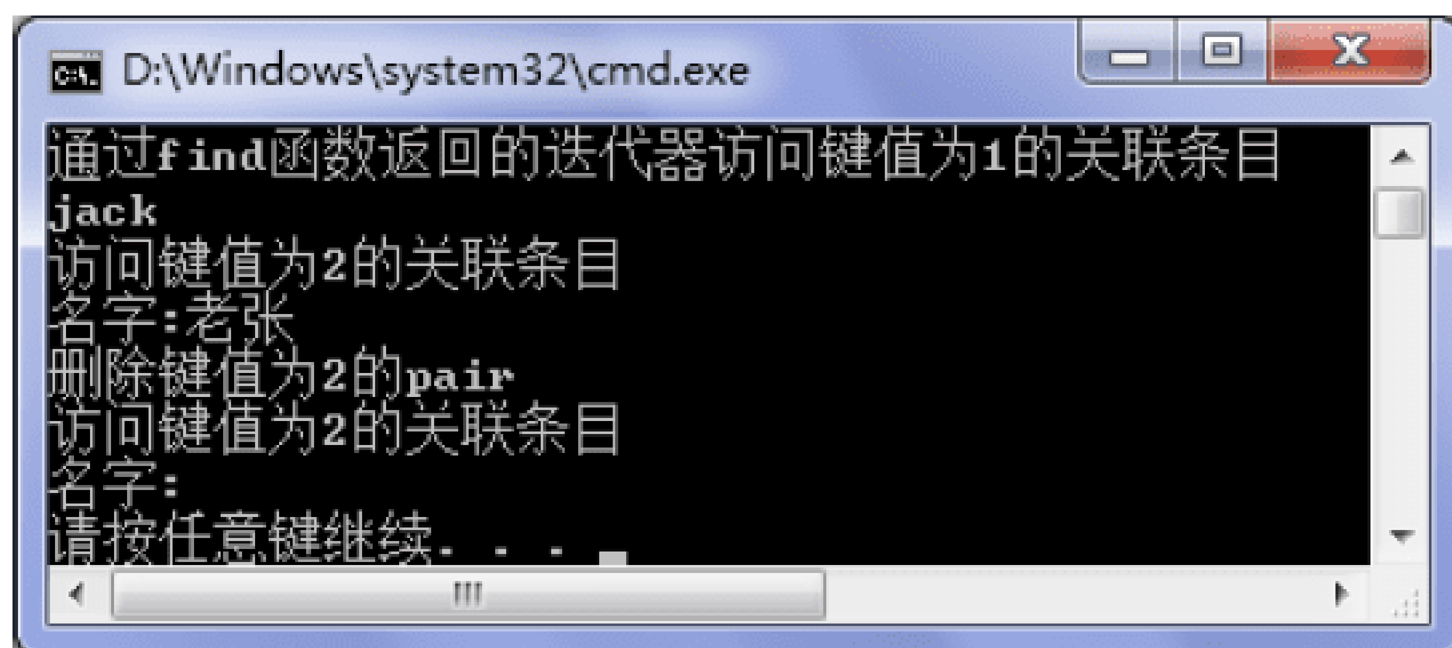


图 15.8　关联容器的操作

15.3　常 用 算 法

算法与程序设计以及数据结构密切相关，是解决一个问题的完整的步骤描述，是解决问题的策略、规则、方法，是求解特定问题的一组有限的操作序列。STL 提供了算法库，算法库中都是模板函数。迭代器主要负责从容器中获取一个对象，算法与具体对象在容器中的什么位置等细节无关。每个算法都是参数化一个或多个迭代器类型的函数模板。

使用 STL 提供的算法需要在程序中包含相应的头文件 algorithm 和 numeric。algorithm 中主要用于容器的操作，numeric 头文件中的算法用来处理数组中的值。下面介绍 STL 几种常用的算法函数。

15.3.1　for_each 函数

```
for_each(first,last,func)
```

对 first 到 last 范围内的各个元素执行函数 func 定义的操作。

【例 15.4】 使用 for_each 算法输出容器内的元素。

实例位置：光盘\MR\Instance\15\15.4

```
#include "stdafx.h"
#include <iostream>
#include <set>
#include <algorithm>
using namespace std;
void Output(int val)
{
	cout << val << ' ';
}
void main()
{
	multiset<int ,less<int> > intSet;
	intSet.insert(7);
	intSet.insert(5);
```



```
        intSet.insert(3);
        cout << "Set:";
        for_each(intSet.begin(),intSet.end(),Output);
        cout << endl;
}
```



程序运行结果如图 15.9 所示。

程序定义 Output 函数用来输出变量值，调用 for_each 算法将 multiset 容器中的值不断地传输给 Output 函数，执行 for_each 算法相当于执行了一个循环语句。



图 15.9　使用 for_each 算法输出容器内的元素

15.3.2　fill 函数

```
fill(first,last,val)
```

把值 val 复制到迭代器 first 和 last 指明范围内的各个元素中。

【例 15.5】 应用 fill 算法对容器元素赋值。

实例位置：光盘\MR\Instance\15\15.5

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void Output(int val)
{
        cout << val << ' ';
}
void main()
{
        vector<int > intVect;
        for(int i=0;i<10;++i)
        intVect.push_back(i);
        cout << "Vect :";
        for_each(intVect.begin(),intVect.end(),Output);
        fill(intVect.begin(),intVect.begin()+5,0);
        cout << endl;
        cout << "Vect :";
        for_each(intVect.begin(),intVect.end(),Output);
        cout << endl;
}
```

程序运行结果如图 15.10 所示。





Note



图 15.10 应用 fill 算法对容器元素赋值

程序向 vector 容器中添加 0～9 共 10 个元素，然后使用 fill 算法将前 5 个元素的值全改为 0，通过在修改前输出全部元素和在修改后输出全部元素，可以观察到 fill 算法的执行效果。

15.3.3 sort 函数

sort(first,last)

对迭代器 first 和 last 指明范围内的元素排序。

【例 15.6】 应用 sort 算法。

实例位置：光盘\MR\Instance\15\15.6

```
#include "stdafx.h"
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
void Output(int val)
{
	cout << val << ' ';
}
void main()
{
	vector<char > charVect;
	charVect.push_back('M');
	charVect.push_back('R');
	charVect.push_back('K');
	charVect.push_back('J');
	charVect.push_back('H');
	charVect.push_back('I');
	cout << "Vect :";
	for_each(charVect.begin(),charVect.end(),Output);
	sort(charVect.begin(),charVect.end());
	cout << endl;
	cout << "Vect :";
	for_each(charVect.begin(),charVect.end(),Output);
	cout << endl;
}
```



程序运行结果如图 15.11 所示。



图 15.11　应用 sort 算法



程序中应用 sort 算法对 vector 容器内的字符元素的 ASCII 码进行递增排序。

15.3.4　transform 函数

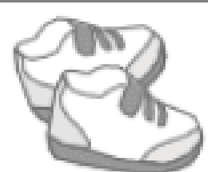
```
transform(first,last,result,func)
```

将指定容器 first 到 last 范围中的元素执行 func 定义的操作，并将返回值依次应用到 result 迭代器所属的容器内。

【例 15.7】 应用 transform 算法。

实例位置：光盘\MR\Instance\15\15.7

```
#include "stdafx.h"
#include <vector>
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
class point
{
public:
	point(){
		x = 0;
		y = 0;
	};
	point(int x,int y){
		this->x = x;
		this->y = y;
	}
	void showPoint()
	{
	cout<<"坐标为("<<x<<","<<y<<")"<<endl;
	}
private:
	int x;
	int y;
};
point squareLine(int x)
```





Note

```
{
	return point(x,x*x);
}
int main()
{
	int array[] =  {1,2,3,4,5,6};
	vector<int> vInt(array,array+6);
	vector<point> vPoint;
	vPoint.resize(6);	//防止越界
	transform(vInt.begin(),vInt.end(),vPoint.begin(),squareLine);
	for(auto it = vPoint.begin();it!=vPoint.end();it++)
	{
		it->showPoint();
	}
	return 0;
}
```

程序运行结果如图 15.12 所示。

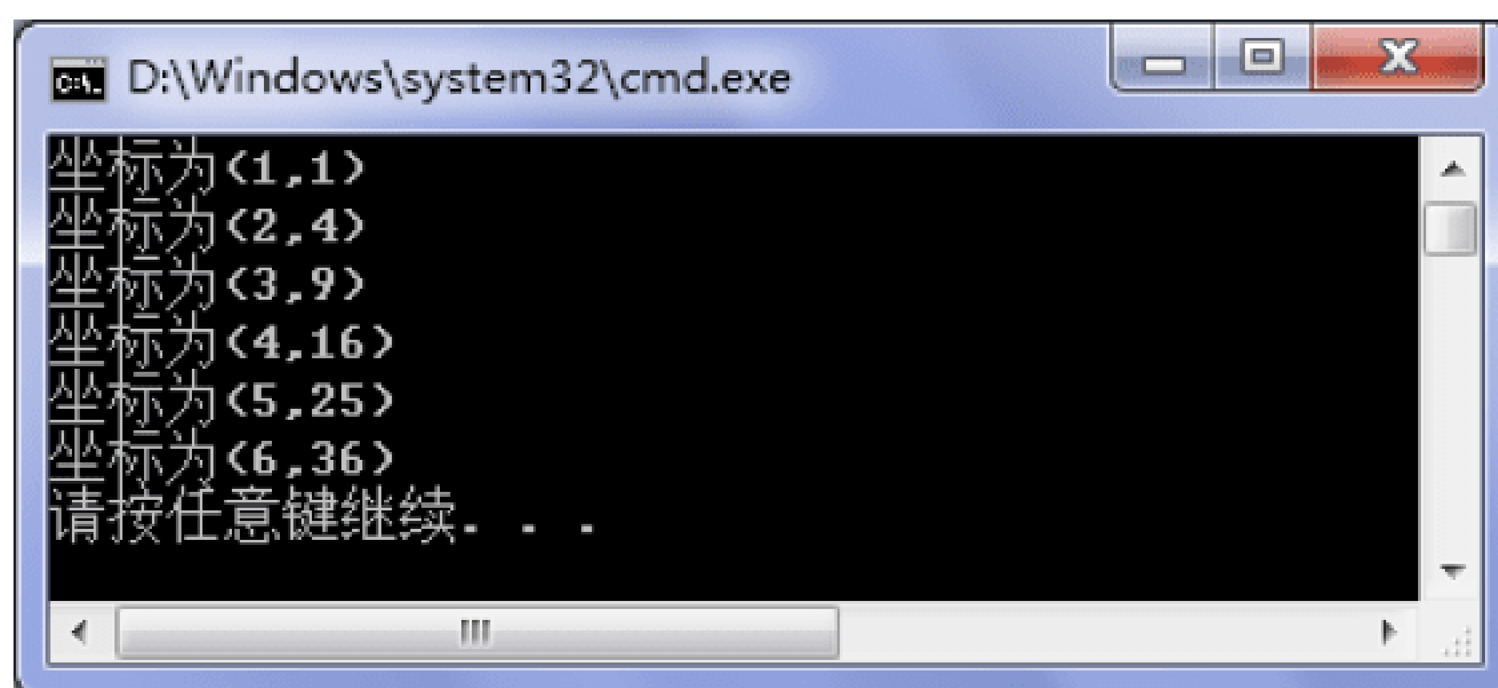


图 15.12　应用 tranform 算法

程序中使用 transform 算法对 vector<int>容器内的所有元素执行了函数 squareLine 的函数操作，使它的返回值添加到 vector<point>容器中。

15.4　lambda 表达式

lambda 表达式是 C++11 所提供的一种新的机制。它不是标准模板库特有的，但广泛应用于此。它实质上是一个匿名函数，下面用一个实例简单说明它的使用方法。

【例 15.8】 lambda 表达式的使用。

实例位置：光盘\MR\Instance\15\15.8

```
#include "stdafx.h"
#include <list>
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
```



```
int main()
{
    int array[3]={1,2,3};
    list<int> list1(array,array+3);
    transform(list1.begin(),list1.end(),list1.begin(),[](int x){return x*x*x;});
    for_each(list1.begin(),list1.end(),[](int x){cout<<x<<endl;});
    return 0;
}
```



程序运行结果如图 15.13 所示。

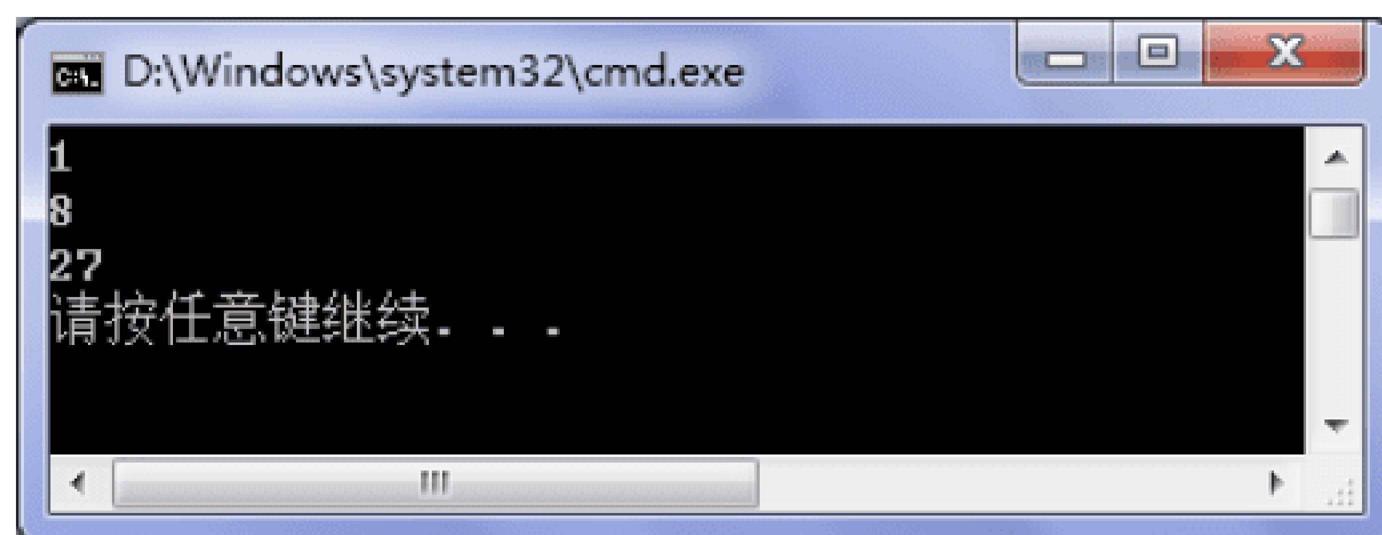


图 15.13 lambda 表达式的使用

代码中的 transform 算法和 for_each 算法分别应用了一个 lambda 表达式，这个表达式的实质即是一个匿名函数。

lambda 表达式的两种类型分别为[](){}和[]()->type{}。

[] 代表从外部传参的约束形式，也称为捕获字句。传递进来的外部参数集合称为闭包。如果修改该外部值传参，需要加上关键字 mutable，如[]()->type{}。

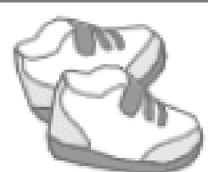
- ☑ []：无约束，同时默认外部值传参。
- ☑ [x, &y]：x 为传递，y 为左值引用传递。
- ☑ [&]：任何外部传参都视为左值引用传递参数。
- ☑ [=]：任何外部传参都视为值传递参数。
- ☑ [&, x]：x 显式地按值传递参数，其他外部参数的是左值引用传递参数。
- ☑ [=, &x]：x 显示地按左值引用传递参数，其他外部参数的是值传递参数。
- ☑ ()：填写声明的形参。
- ☑ type：为返回值。
- ☑ {}：中填写处理数据语句，若需要返回值则填写 return。若使用第一种表达式，语句一定要是单句。否则，按第二种方式填写返回值类型。

下面用一个实例来演示闭包的应用。

【例 15.9】 闭包的应用。

实例位置：光盘\MR\Instance\15\15.9

```
#include "stdafx.h"
#include <vector>
#include <map>
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
int main()
{
    cout<<"检查小于 60 的数，记录下它们的值，之后将它们替换成 60"<<endl;
    int array[10]={70,89,77,30,61,47,55,21,67,31};
    map<int,int> record1;
```






```
    map<int,int> record2;
    vector<int> vInt1(array,array+10);
    vector<int> vInt2(array,array+10);
    int index1 = 0;
    int index2 = 0;
    transform(vInt1.begin(),vInt1.end(),vInt1.begin(),
    [&](int x)->int{
    if(x<60)
    {
        pair<int,int>temp(index1,x);
        record1.insert(temp);
        x=60;
        index1++;
    }
    return x;});
    transform(vInt2.begin(),vInt2.end(),vInt2.begin(),
    [&,record2](int x)mutable->int{
    if(x<60)
    {
        pair<int,int> temp(index2,x);
        record2.insert(temp);
        index2++;
        x=60;
    }
    return x;});
    if(record1.empty())
    {
        cout<<"record1 是空的"<<endl;
    }
    else
    {
        cout<<"record1 不是空的，它的各项值:"<<endl;
        for_each(record1.begin(),record1.end(),[](pair<int,int>p)
        {cout<<p.second<<" ";});
    }
    cout<<endl;
    if(record2.empty())
    {
        cout<<"record2 是空的"<<endl;
    }
    cout<<"输出 vInt1 中的值"<<endl;
    for_each(vInt1.begin(),vInt1.end(),[](int x){cout<<x<<" ";});
    cout<<endl;
    cout<<"输出 vInt2 中的值"<<endl;
    for_each(vInt2.begin(),vInt2.end(),[](int x){cout<<x<<" ";});
    cout<<endl;
    return 0;
}
```

程序运行结果如图 15.14 所示。

程序中对两个 map 模板对象使用了两种外部传参约束，结果为 record1 记录了数据，record2



没有改变。这与函数传递参数的规律完全符合。

lambda 表达式是一个匿名表达式，通过函数名调用它是不可能的。但是 C++11 提供了包装它的方法。



【例 15.10】 匿名函数 lambda 表达式的包装。

实例位置：光盘\MR\Instance\15\15.10

```
#include "stdafx.h"
#include <iostream>
#include <list>
#include <algorithm>
#include <functional>        //提供了包装 lambda 的函数模板
using namespace std;
int main()
{
      function<int(int)>hcf = [&](int x)->int{;return x*x*x;};
      int p=5;
      int a = hcf(p)
      //hcf 变成了犹如 int(x){k2=123;return k1=x*x*x}这样一个临时函数，很方便地调用 main 块中
      //的变量。在其他的情况下，如成员函数，全局函数中都可很便利地应用
      cout<<"输出 p 的三次方"<<a<<endl;
      return 0;
}
```

程序运行结果如图 15.15 所示。



图 15.14　闭包的应用



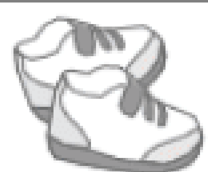
图 15.15　lambda 表达式的包装

程序中首先包含了头文件 functional.h，它提供了包装 lambda 表达式的模板。之后使用 function<T(T1)>模板 hcf 对 lambda 表达式进行了包装。模板中的类型 T 代表返回值，T1 代表形参列表类型。当形参为多个时，function 模板的形参列表也可以做相应的扩展。

15.5　综 合 应 用

15.5.1　迭代输出信息

【例 15.11】 建立一个整数的 vector 标准容器，装入一些 int 类型元素，使用 sort 算法，根



据选用迭代器的起始位置，将这些元素按升序排列。关键代码如下：



实例位置：光盘\MR\Instance\15\15.11

```
vector<int> l;
l.push_back(2231);
l.push_back(1456);
l.push_back(789);
l.push_back(11);
sort(l.begin(),l.end());
```

程序运行结果如图 15.16 所示。



图 15.16 迭代输出信息

15.5.2 计算平均值

【例 15.12】 本例使用一个 vector 标准容器储存浮点数的值，使用 transform 算法和 lambda 表达式计算每一元素位置之前的所有元素的平均值，将这些值保存到另外一个 vector 容器中。关键代码如下：

实例位置：光盘\MR\Instance\15\15.12

```
int a[5]={1.11f,3.33f,2.22f,6.66f,13.78f};
vector<float> init(a,a+5);
vector<float> result;
result.resize(5);
float tmpAdv = 0.0f;
float index = 1.0f;
transform(init.begin(),init.end(),result.begin(),[&](floatf)->float{tmpAdv +=f;
tmpAdv = tmpAdv/index;index++;return tmpAdv;});
```

程序运行结果如图 15.17 所示。

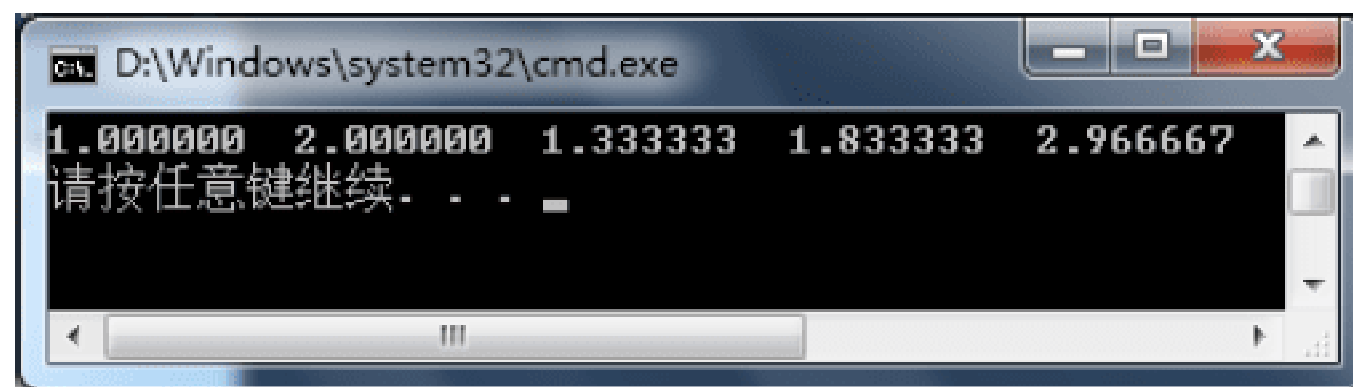


图 15.17 计算平均值

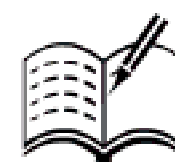
15.6 本章常见错误

15.6.1 不要直接使用头指针操作链表

对一个链表的增加节点、删除节点等操作通常是通过链表头来实现的，由头指针向后依次引



用想要修改的节点。但是最后定义一个临时指针来代替头指针去操作链表。这样能保护头指针。例如遍历输出链表，如果直接用头指针去遍历，最后头指针指向 NULL，就无法再找到这个链表了。



Note

15.6.2 区分内存中的栈和数据结构中的栈

内存管理中的栈（stack）是内存中的一个区域。内存分为 5 个区，分别是栈、堆、静态全局区、文字常量区和代码区。栈区存储局部变量，系统自动给变量分配空间。这里的栈采用的就是数据结构中的栈的思想，即遵循先进后出的管理方法。

数据结构中的栈，是一种遵循先进后出的数据结构。在系统内存中，堆区采用堆的数据结构管理，栈区是采用栈的数据结构管理。

15.6.3 数组和容器的区别

数组是容器的一种。容器可以理解为一个类，数组理解为一个对象。数组、链表等都是一种容器。数组里可以存放类型相同的数据，在内存中占据一个连续的存储空间。但是数组一旦定义，大小就固定了，容器可以根据存储数据大小自动调整大小。

15.7 本章小结

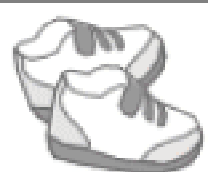
本章主要介绍了标准模板库中的容器、算法和迭代器，这三者是标准模板库的核心内容，并且相互联系非常密切。迭代器是访问容器中的元素，算法是对容器中的元素进行操作。每种容器都有各自的特点，只有熟练掌握这些特点才能将标准模板库的作用充分发挥。lambda 表达式与标准模板库搭配使用使得匿名函数的定义与外部传参都很方便。

15.8 跟我上机

参考答案：光盘\MR\跟我上机

用数组模拟压栈和弹栈。压栈过程向数组输入数据，超过数组容量代表栈满；弹栈过程从数组输出数据，没有数据可输出时代表栈空。实现如下：

```
#include "stdafx.h"
#include <iostream>
#define MAXSIZE 50                    //数组容量
using std::cout;
```





```
using std::endl;
//压栈
bool push(int *stack,int PushX,int &top)
{
	if(MAXSIZE == top)			//判断是否栈满
	{
		cout<<"栈满"<<endl;		//如果栈满提示信息，退出压栈函数
		return false;			//压栈失败返回 false
	}
	else
	{
		stack[top++] = PushX;		//将数据压入 top 标识的位置，top 向上移动一个位置
		return true;			//压栈成功返回 true
	}
}
//弹栈
bool pop(int *stack,int &X,int &top)
{
	if(!top)				//栈顶标识为 0，提示栈空，退出函数
	{
		cout<<"栈空"<<endl;
		return false;			//弹栈失败返回 false
	}
	else
	{
		X = stack[--top];		//弹出栈顶元素
		return true;			//弹栈成功返回 true
	}
}
int _tmain(int argc, _TCHAR* argv[])
{
	int stack[MAXSIZE]={0};		//定义整型数组模拟栈结构
	int top = 0;			//定义栈顶标识
	int X = 10;
	for(int i=0;i<3;i++)
		if(push(stack,X++,top))	//压栈，压入 10，11，12
			cout<<"压入元素"<<stack[i]<<endl;
	cout<<"共压入"<<top<<"个元素\n"<<endl;
	for(int i=0;i<3;i++)
		if(pop(stack,X,top))
			cout<<"弹出元素"<<X<<endl;
	return 0;
}
```

第16章

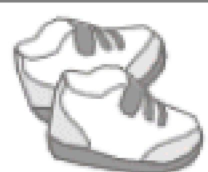
利用文件处理数据

（视频讲解：58 分钟）

文件操作是程序开发中不可缺少的一部分，任何需要数据存储的软件都需要进行文件操作。文件操作包括打开文件、读文件和写文件，掌握读文件和写文件的同时，还要理解文件指针的移动，这能够控制读文件和写文件的位置。

本章能够完成的主要范例（已掌握的在方框中打勾）

- ☐ 创建并打开一个文件
- ☐ 使用 ifstream 和 ofstream 对象实现读写文件的功能
- ☐ 向文本文件写入数据
- ☐ 读取文本文件内容
- ☐ 使用 read 读取二进制文件
- ☐ 实现文件复制
- ☐ 追加写入文件
- ☐ 记录并输出空格位置



16.1　文件流概述



Note

16.1.1　C++中的流类库

C++语言中为不同类型数据的标准输入和输出定义了专门的类库，类库中主要有 ios、istream、ostream、iostream、ifstream、ofstream、fstream、istrstream、ostrstream 和 strstream 等类。ios 为根基类，它直接派生 4 个类，输入流类 istream、输出流类 ostream、文件流基类 fstreambase 和字符串流基类 strstreambase。输入文件流类 ifstream 同时继承了输入流类和文件流基类，输出文件流类 ofstream 同时继承了输出流类和文件流基类，输入字符串流类 istrstream 同时继承了输入流类和字符串流基类，输出字符串流类 ostrstream 同时继承了输出流类和字符串流基类，输入/输出流类 iostream 同时继承了输入流类和输出流类，输入/输出文件流类 fstream 同时继承了输入/输出流类和文件流基类，输入/输出字符串流类 strstream 同时继承了输入/输出流类和字符串流基类。类库关系如图 16.1 所示。

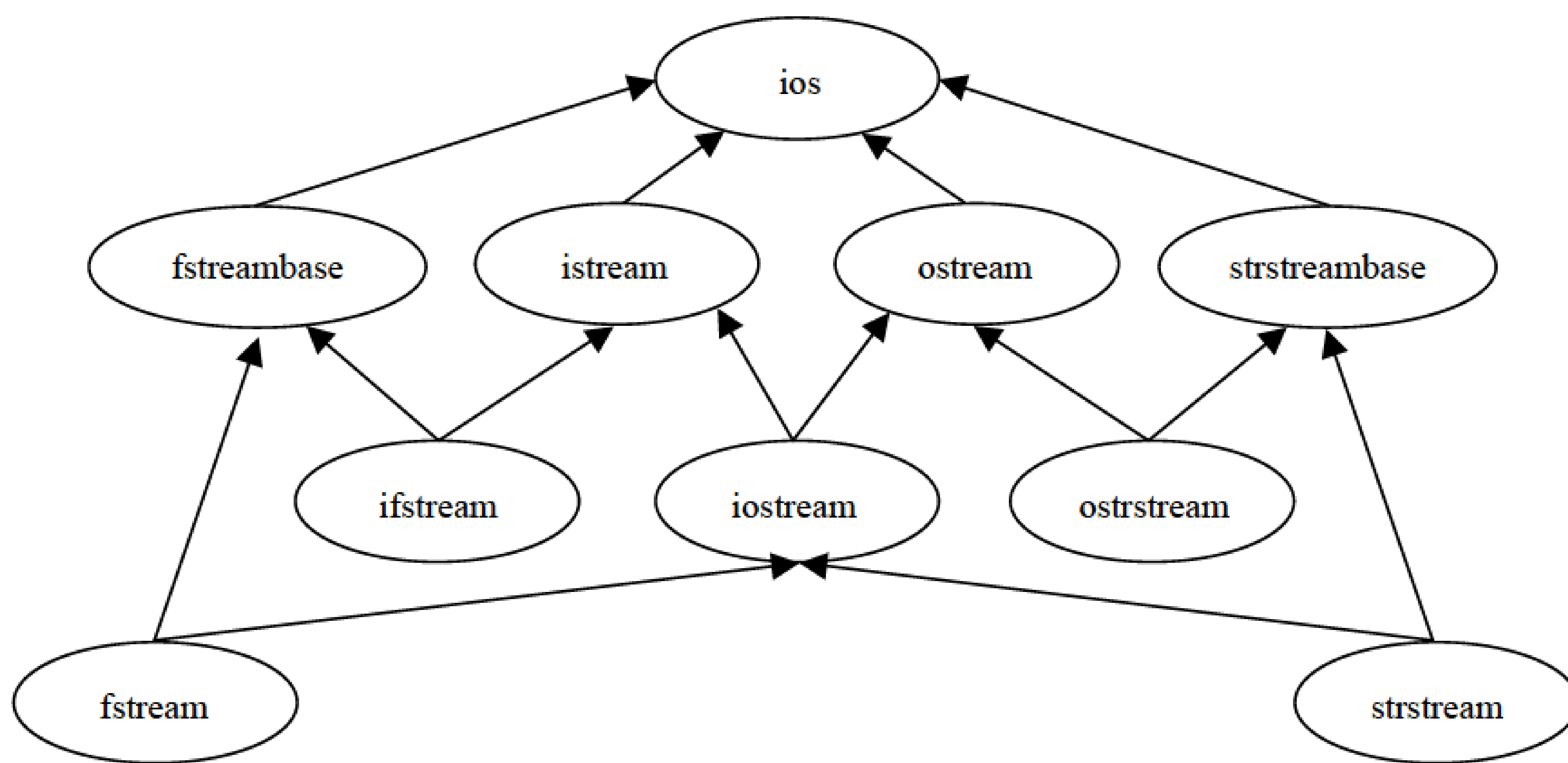


图 16.1　类库关系图

16.1.2　使用类库

C++系统中的 I/O 标准类，都定义在 iostream、fstream 和 strstream 这 3 个头文件中，各头文件包含的类如下：

☑　进行标准 I/O 操作时使用 iostream 头文件，它包含 ios、iostream、istream 和 ostream 等类。

☑　进行文件 I/O 操作时使用 fstream 头文件，它包含 fstream、ifstream、ofstream 和 fstreambase 等类。

☑　进行串 I/O 操作时使用 strstream 头文件，它包含 strstream、istrstream、ostrstream、strstreambase 和 iostream 等类。



要进行什么样的操作，只要引入头文件就可以使用类进行操作了。

16.1.3　ios 类中的枚举常量



Note

在根基类 ios 中定义了用户需要使用的枚举类型，由于它们是在公用成员部分定义的，所以其中的每个枚举类型常量在加上 ios::前缀后都可以被本类成员函数和所有外部函数访问。

在 3 个枚举类型中有一个无名枚举类型，其中定义的每个枚举常量都是用于设置控制输入/输出格式的标志的。该枚举类型定义如下：

```
enum{skipws,left,right,insternal,dec,oct,hex,showbase,showpoint,uppercase,showpos,scientific,fixed,unitbuf,
stdio};
```

主要枚举常量的含义如下。

- ☑ skipws：利用它设置对应标志后，从流中输入数据时跳过当前位置及后面的所有连续的空白字符，从第一个非空白字符起读数，否则不跳过空白字符。空格、制表符\t、回车符\r 和换行符\n 统称为空白符。默认为设置。
- ☑ left：靠左对齐输出数据。
- ☑ right：靠右对齐输出数据。
- ☑ insternal：显示占满整个域宽，用填充字符在符号和数值之间填充。
- ☑ dec：用十进制输出数据。
- ☑ hex：用十六进制输出数据。
- ☑ showbase：在数值前显示基数符，八进制基数符是 0，十六进制基数符是 0x。
- ☑ showpoint：强制输出的浮点数中带有小数点和小数尾部的无效数字 0。
- ☑ uppercase：用大写输出数据。
- ☑ showpos：在数值前显示符号。
- ☑ scientific：用科学计数法显示浮点数。
- ☑ fixed：用固定小数点位数显示浮点数。

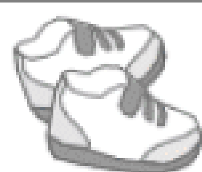
16.1.4　使用流进行输出

通过前文的学习，相信读者已经对文件流有了一定的了解，现在就通过实例来看一下如何在程序中使用流进行输出。

【例 16.1】 字符相加。

实例位置：光盘\MR\Instance\16\16.1

```
#include "stdafx.h"
#include <iostream>
#include <strstream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
```





Note

```
{
    char buf[]="12345678";
    int i,j;
    istrstream s1(buf);
    s1 >> i;                        //将字符串转换为数字
    istrstream s2(buf,3);
    s2 >> j;                        //将字符串转换为数字
    cout << i+j <<endl;             //两个数字相加
}
```

程序运行结果如图 16.2 所示。

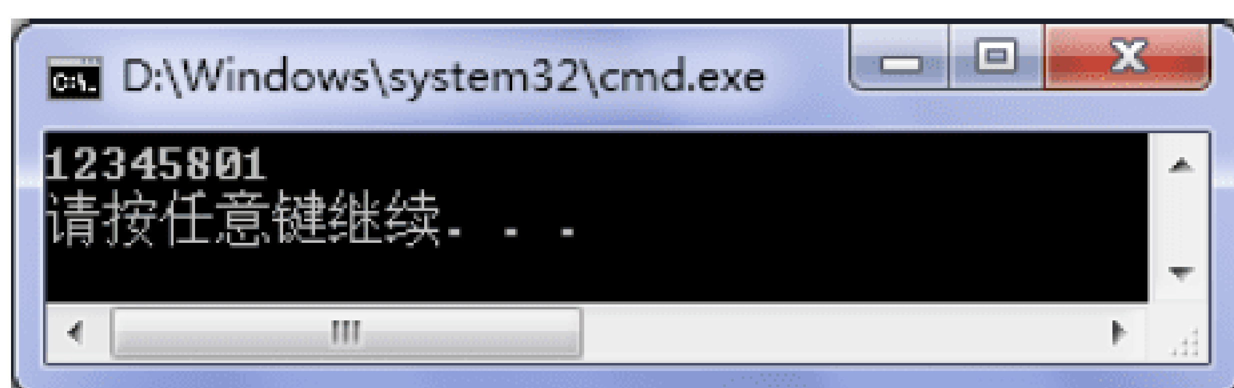


图 16.2　字符相加

16.2　打 开 文 件

16.2.1　文件打开方式

只有使用文件流与磁盘上的文件进行连接后才能对磁盘上的文件进行操作，这个连接过程称为打开文件。

打开文件的方式有以下两种。

（1）在创建文件流时利用构造函数打开文件，即在创建流时加入参数，语法结构如下：

```
<文件流类> <文件流对象名>(<文件名>,<打开方式>)
```

其中文件流类可以是 fstream、ifstream 和 ofstream 中的一种。文件名指的是磁盘文件的名称，包括磁盘文件的路径名。打开方式在 ios 类中定义，有输入方式、输出方式、追加方式等。

☑　ios::in：用输入方式打开文件，文件只能读取，不能改写。

☑　ios::out：以输出方式打开文件，只能改写，不能读取。

☑　ios::app：以追加方式打开文件，打开后文件指针在文件尾部，可改写。

☑　ios::ate：打开已存在的文件，文件指针指向文件尾部，可读可写。

☑　ios::binary：以二进制方式打开文件。

☑　ios::trunc：打开文件进行写操作，如果文件已经存在，清除文件中的数据。

☑　ios::nocreate：打开已经存在的文件，如果文件不存在，打开失败，不创建。

☑　ios::noreplace：创建新文件，如果文件已经存在，打开失败，不覆盖。

参数可以结合运算符|使用，如下所示。





☑　ios::in|ios::out：以读写方式打开文件，对文件可读可写。

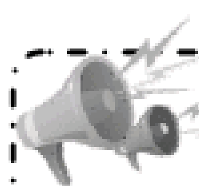
☑　ios::in|ios::binary：以二进制方式打开文件，进行读操作。

使用相对路径打开文件 test.txt 进行写操作：

```
ofstream outfile("test.txt",ios::out);
```

使用绝对路径打开文件 test.txt 进行写操作：

```
ofstream outfile("c:\\test.txt",ios::out);
```



注意：

字符\表示转义，如果使用 c:\则必须写成 c:\\。

（2）利用 open 函数打开磁盘文件，语法结构如下：

```
<文件流对象名>.open(<文件名>,<打开方式>);
```

文件流对象名是一个已经定义了的文件流对象：

```
ifstream infile;
infile.open("test.txt",ios::out);
```

使用两种方式中的任意一种打开文件后，如果打开成功，文件流对象为 0 值，如果打开失败，则文件流对象为非 0 值。检测一个文件是否打开成功可以用以下语句：

```
void open(const char * filename,int mode,int prot=filebuf::openprot)
```

prot 决定文件的访问方式，取值说明如下。

☑　0：普通文件。

☑　1：只读文件。

☑　2：隐含文件。

☑　4：系统文件。

16.2.2　默认打开模式

如果没有指定打开方式参数，编译器会使用默认值。

```
std::ofstream std::ios::out | std::ios::trunk
std::ifstream std::ios::in
std::fstream  无默认值
```

文件打开模式根据用户的需要有不同的组合，下面就对各个模式的效果进行介绍。文件打开模式如表 16.1 所示。

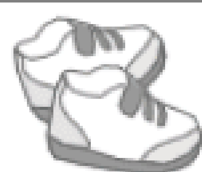




表 16.1 文件打开模式

打开方式	效果	文件存在	文件不存在
in	为读而打开		错误
out	为写而打开	截断	创建
out | trunc			
out | app	为在文件结尾处写而打开		创建
in | out	为输入/输出而打开		创建
in | out | trunc	为输入/输出而打开	截断	创建

16.2.3 创建并打开文件

通过前文的学习，相信读者已经对文件操作的知识有了一定的了解。为了使读者更好地掌握前面学习的知识，下面通过实例进一步介绍。

【例 16.2】 创建文件。

实例位置：光盘\MR\Instance\16\16.2

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
	ofstream ofile;                          //定义一个文件对象
	cout << "Create file1" << endl;
	ofile.open("test.txt");                  //调用 open 函数，打开 test.txt 文件
	if(!ofile.fail())                        //出错判断
	{
		ofile << "name1" << " ";
		ofile << "sex1" << " ";
		ofile << "age1";
		ofile.close();
		cout << "Create file2" <<endl;
		ofile.open("test2.txt");
		if(!ofile.fail())
		{
			ofile << "name2" << " ";
			ofile << "sex2" << " ";
			ofile << "age2";
			ofile.close();
		}
	}
	return 0;
}
```

程序执行后将会在项目目录下创建两个文件，如图 16.3、图 16.4 和图 16.5 所示。





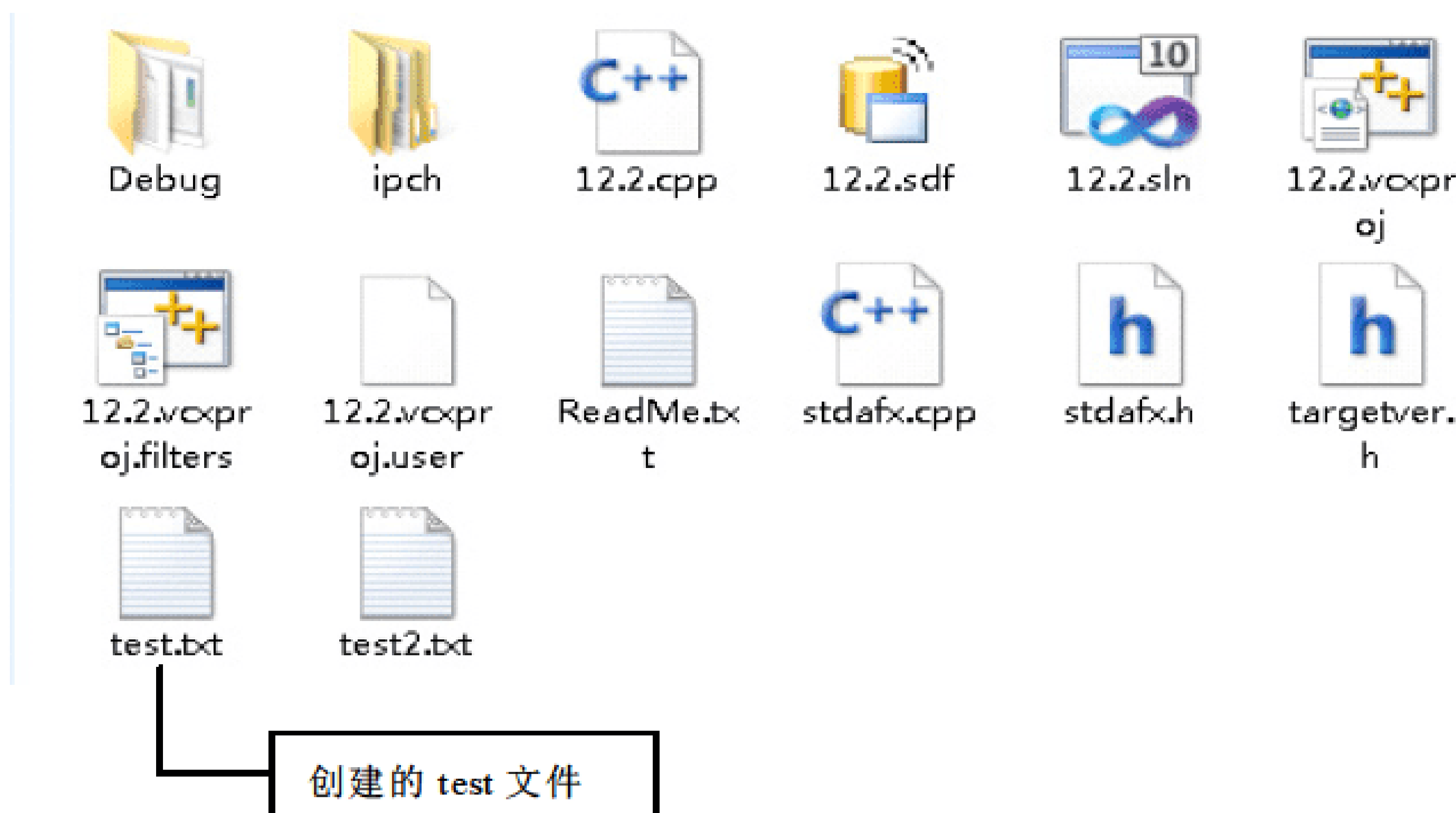


图 16.3　创建的文件

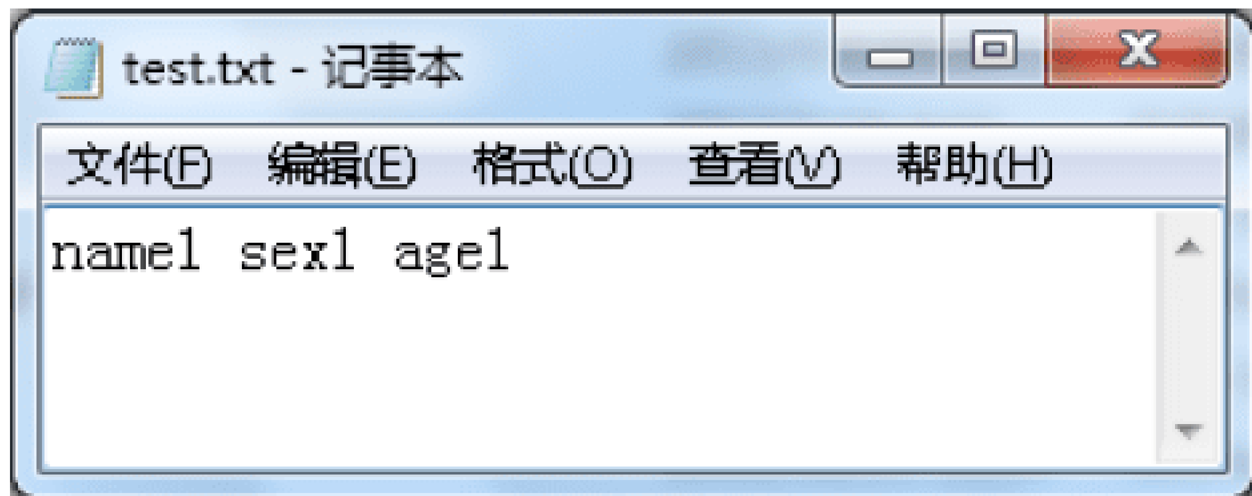


图 16.4　text 文件内容

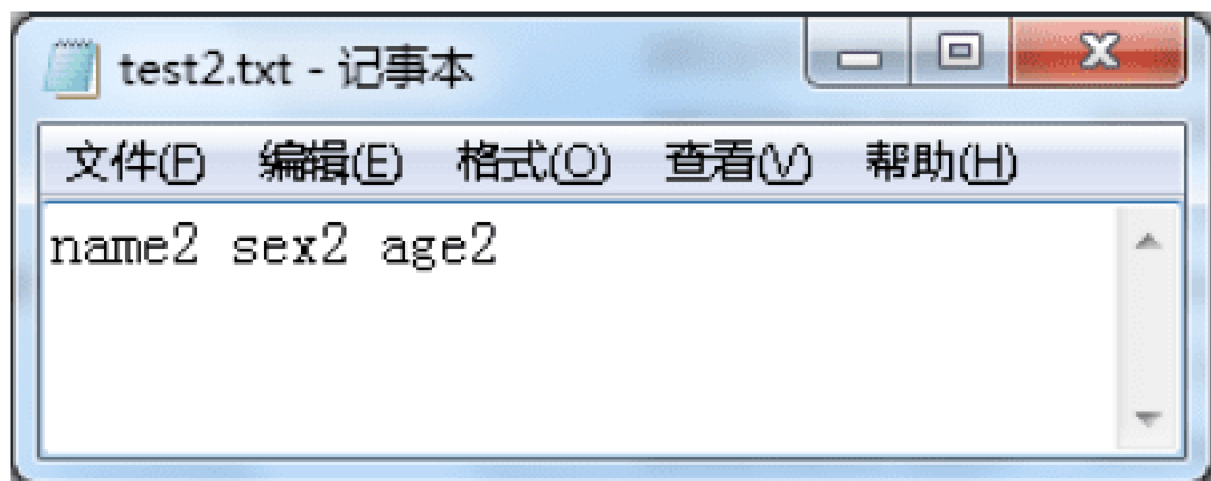


图 16.5　text2 文件内容

程序运行将会创建两个文件，由于 ofstream 默认打开方式是 std::ios::out | std::ios::trunk，所以当文件夹内没有 test.txt 文件和 test2.txt 文件时，会创建这两个文件，并向文件写入字符串。向 test.txt 文件写入字符串“name1 sex1 age1”，向 test2.txt 文件写入字符串“name2 sex2 age2”。如果文件夹内有 test.txt 文件和 test2.txt 文件时，程序会覆盖原有文件而重新写入。

16.3　读 写 文 件

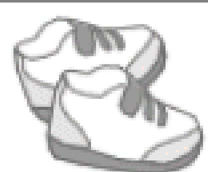
在对文件进行操作时，必然离不开读写文件。在使用程序查看文件内容时，首先要读取文件，而要修改文件内容时，则需要向文件中写入数据，本节主要介绍通过程序对文件的读写操作。

16.3.1　文件流分类

流可以分为 3 类，即输入流、输出流和输入/输出流，相应地必须将流说明为 ifstream、ofstream 和 fstream 类的对象。

```
ifstream ifile;                    //声明一个输入流
ofstream ofile;                    //声明一个输出流
fstream iofile;                    //声明一个输入/输出流
```

说明了流对象之后，可以使用函数 open()打开文件。文件的打开即是在流与文件之间建立一



个连接。

ofstream 和 ifstream 类有很多用于磁盘文件管理的函数。

☑ attach：在一个打开的文件与流之间建立连接。
☑ close：刷新未保存的数据后关闭文件。
☑ flush：刷新流。
☑ open：打开一个文件并把它与流连接。
☑ put：把一个字节写入流中。
☑ rdbuf：返回与流连接的 filebuf 对象。
☑ seekp：设置流文件指针位置。
☑ setmode：设置流为二进制或文本模式。
☑ tellp：获取流文件指针位置。
☑ write：把一组字节写入流中。

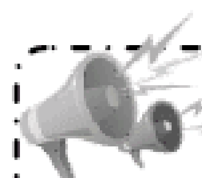


Note

fstream 成员函数如表 16.2 所示。

表 16.2　fstream 成员函数

函　数　名	功 能 描 述
get(c)	从文件读取一个字符
getline(str,n, '\n')	从文件读取字符存入字符串 str 中，直到读取 n-1 个字符或遇到\n 时结束
peek()	查找下一个字符，但不从文件中取出
put(c)	将一个字符写入文件
putback(c)	对输入流返回一个字符，但不保存
eof	如果读取超过 eof，返回 true
ignore(n)	跳过 n 个字符，参数为空时，表示跳过下一个字符



注意：

表中参数 c、str 为 char 型，参数 n 为 int 型。

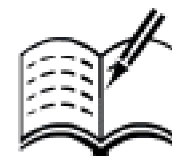
通过上面的介绍，读者已经对写入流有了一定的了解，下面就通过使用 ifstream 和 ofstream 对象实现读写文件的功能。

【例 16.3】 使用 ifstream 和 ofstream 对象实现读写文件的功能。

实例位置：光盘\MR\Instance\16\16.3

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
	char buf[128];
	ofstream ofile("test.txt");                    //打开文件
	cout<<"请输入文字：\n";
	for(int i=0;i<5;i++)
```





Note

```
    {
        memset(buf,0,128);                          //清空数组
        cin >> buf;                                 //给数组赋值
        ofile << buf;                               //写入文件
    }
    ofile.close();                                  //关闭文件
    ifstream ifile("test.txt");
    while(!ifile.eof())                             //文件打开成功则执行
    {
        char ch;
        ifile.get(ch);                              //从文件中获取字符存入字符变量 ch 中
        if(!ifile.eof())
            cout << ch;
    }
    cout << endl;
    ifile.close();
    return 0;
}
```

程序运行结果如图 16.6 所示。

程序首先使用 ofstream 类创建并打开了项目目录下的 test.txt 文件，然后需要用户输入 5 次数据，程序把这 5 次输入的数据全部写入 test.txt 文件，接着关闭 ofstream 类打开的文件，用 ifstream 类打开文件，将文件中的内容输出。

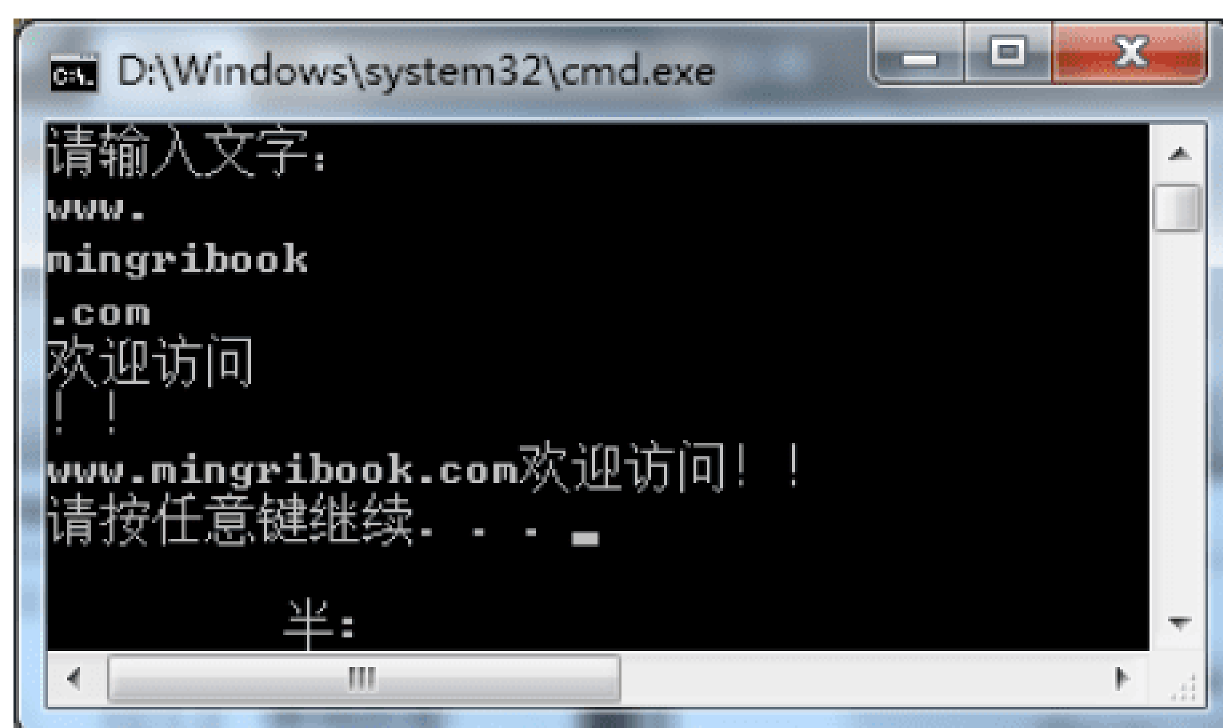


图 16.6　运行结果

16.3.2　写文本文件

文本文件是程序开发经常用到的文件，使用记事本程序就可以打开文本文件。文本文件以.txt 作为扩展名，16.3.1 节已经使用 ifstream 和 ofstream 类创建并写入了文本文件，本节主要应用 fstream 写入文本文件。

【例 16.4】 向文本文件写入数据。

实例位置：光盘\MR\Instance\16\16.4

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    fstream file("test.txt",ios::out);
    if(!file.fail())
    {
        cout << "start write " << endl;
        file << "name" << " ";
```


Note

```
            file << "sex" << " ";
            file << "age" << endl;
            file << "张三" << " ";
            file << "男" << " ";
            file << "26" << endl;
        }
        else
            cout << "can not open" << endl;
        file.close();
        return 0;
}
```

程序运行结果如图 16.7 所示。

程序通过 fstream 类的构造函数打开文本文件 test.txt，然后向文本文件写入了“name sex age”，换行输入了“张三 男 26”。

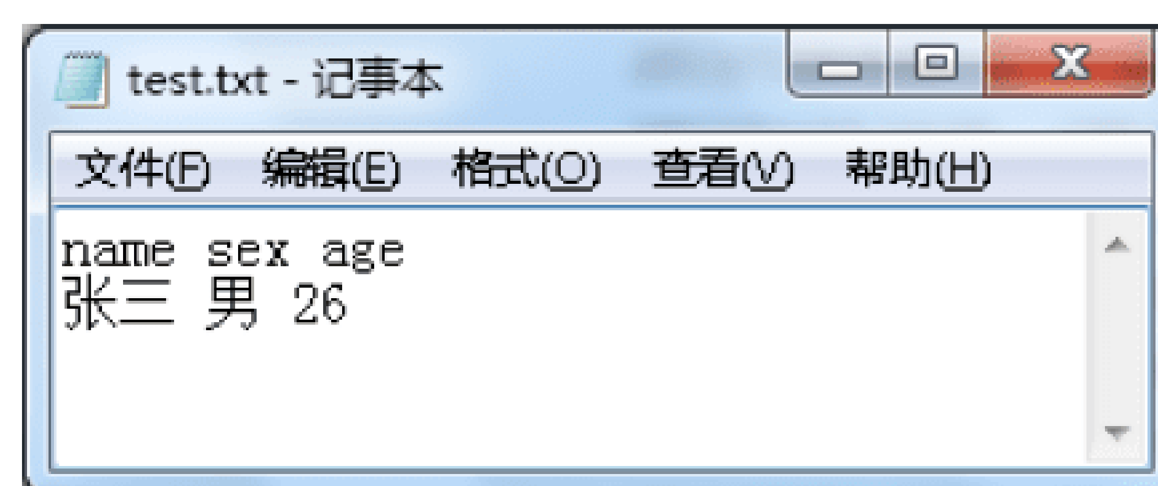


图 16.7 写入文件的内容

16.3.3 读取文本文件

前面介绍了如何写入文件信息，下面通过实例来介绍如何读取文本文件的内容。

【例 16.5】 读取文本文件内容。

实例位置：光盘\MR\Instance\16\16.5

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
        fstream file("古诗.txt",ios::in);
        if(!file.fail())
        {
            while(!file.eof())
            {
                char buf[128];
                file.getline(buf,128);
                if(file.tellg()>0)
                {
                    cout << buf;
                    cout << endl;
                }
            }
        }
        else
            cout << "can not open" << endl;;
        file.close();
        return 0;
}
```

程序打开文本文件 test.txt，文件的内容如图 16.8 所示。





Note

程序读取文本文件 test.txt 中的内容，并将其输出，运行结果如图 16.9 所示。

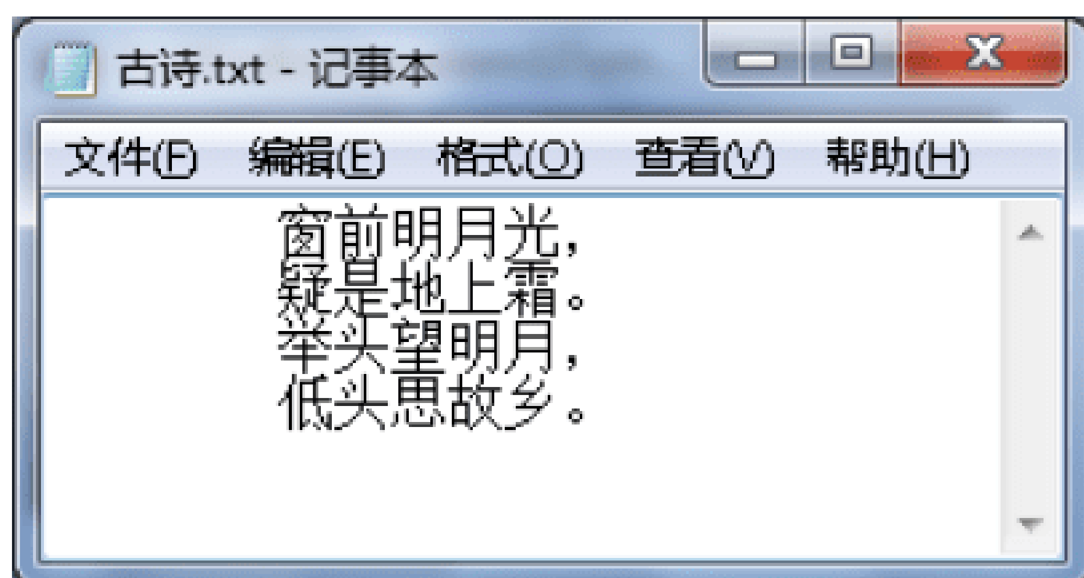


图 16.8　文本文件内容

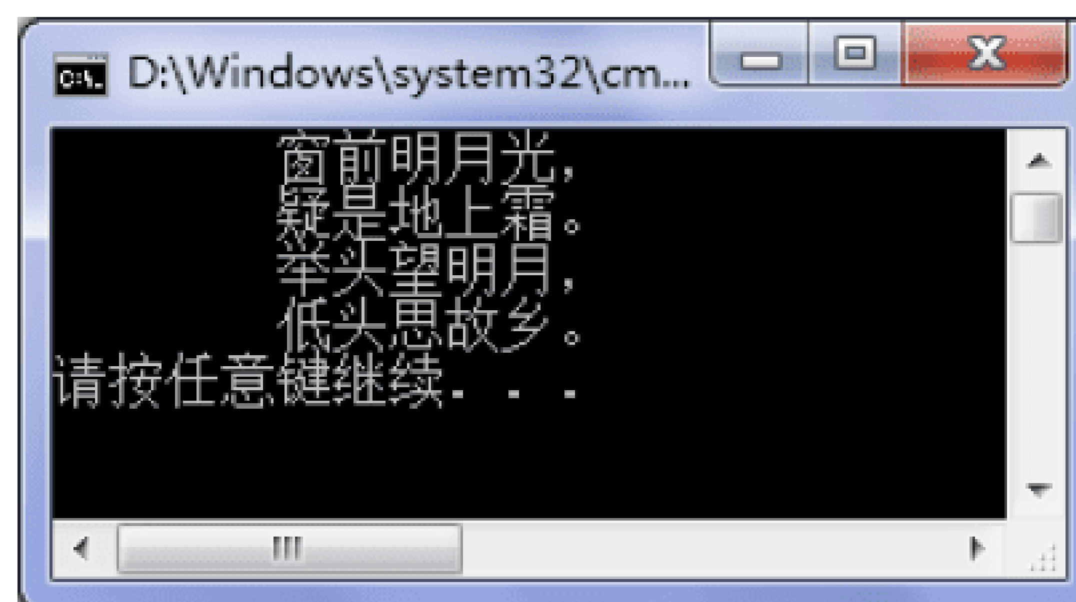


图 16.9　读取文本文件

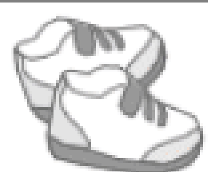
16.3.4　二进制文件的读写

文本文件中的数据都是 ASCII 码，如果要读取图片的内容，就不能使用读取文本文件的方法了。以二进制方式读写文件，需要使用 ios::binary 模式，下面通过实例来实现这一功能。

【例 16.6】 使用 read 读取文件。

实例位置：光盘\MR\Instance\16\16.6

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
        char buf[50];
        fstream file;
        file.open("test.",ios::binary|ios::out);
        for(int i=0;i<2;i++)
        {
                memset(buf,0,50);
                cin >> buf;
                file.write(buf,50);
                file << endl;
        }
        file.close();
        file.open("test.dat",ios::binary|ios::in);
        while(!file.eof())
        {
                memset(buf,0,50);
                file.read(buf,50);
                if(file.tellg()>0)
                        cout << buf;
        }
        cout << endl;
        file.close();
        return 0;
}
```





程序运行结果如图 16.10 所示。



图 16.10　写入与读取文件

程序需要用户输入两次数据，然后通过 fstream 以二进制方式写入到文件，再通过 fstream 以二进制方式读取出来并输出。对二进制数据读取需要使用 read 方法，写入二进制数据需要使用 write 方法。



说明：

cout 遇到结束符\0 就停止输出。在以二进制存储数据的文件中会有很多结束符\0，遇到结束。

16.3.5　实现文件复制

用户在进行程序开发时，有时需要用到复制等操作，下面就来介绍复制文件的方法。

【例 16.7】 文件复制。

实例位置：光盘\MR\Instance\16\16.7

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
	ifstream infile;
	ofstream outfile;
	char name1[20],name2[20];
	char c;
	cout<<"请输入文件："<<"\n";
	cin>>name1;
	infile.open(name1);
	if(!infile)
	{
		cout<<"文件打开失败！";
		exit(1);
	}
	strcpy(name2, "复本");
	strcat(name2,name1);
```





```
        cout<< "start copy" << endl;
        outfile.open(name2);
        if(!outfile)
        {
            cout<<"无法复制";
            exit(1);
        }
        while(infile.get(c))
        {
            outfile << c;
        }
        cout<<"start end"<< endl;
        infile.close();
        outfile.close();
        return 0;
}
```

程序需要用户输入一个文件名，然后使用 infile 打开文件，接着在文件名后加上“复本”两个字，并用 outfile 创建该文件，然后通过一个循环将原文件中的内容复制到目标文件内，完成文件的复制。运行程序，结果如图 16.11 和图 16.12 所示。



图 16.11　文件复制操作

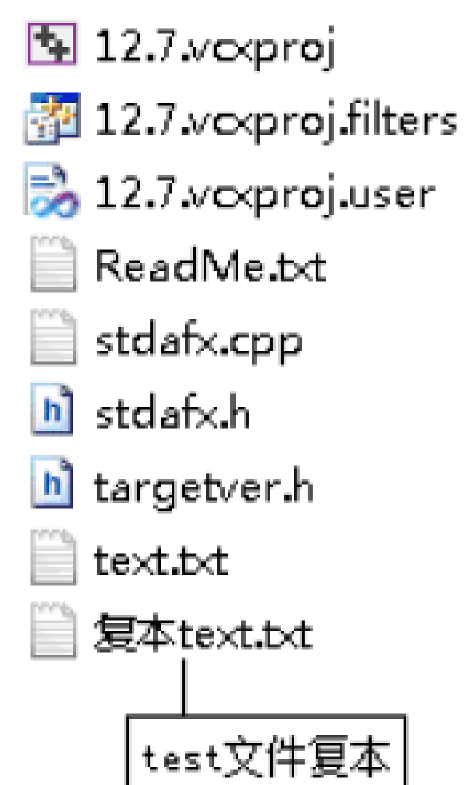


图 16.12　文件的复本

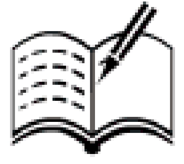
16.4　移动文件指针

在读写文件的过程中，有时用户可能不需要对整个文件进行读写，而是对指定位置的一段数据进行读写操作，这时就需要通过移动文件指针来完成。

16.4.1　文件错误与状态

在 I/O 流的操作过程中可能出现各种错误，每一个流都有一个状态标志字，以指示是否发生了错误及出现了哪种类型的错误，这种处理技术与格式控制标志字是相同的。ios 类定义了以下枚举类型：





Note

```
enum io_state
{
	goodbit=0x00,			//不设置任何位，一切正常
	eofbit=0x01,			//输入流已经结束，无字符可读入
	failbit=0x02,			//上次读/写操作失败，但流仍可使用
	badbit=0x04,			//视图进行无效的读/写操作，流不再可用
	bardfail=0x80			//不可恢复的严重错误
};
```

对应于标志字各状态位，ios 类还提供了以下成员函数来检测或设置流的状态。

```
int rdstate();
int eof();
int fail();
int bad();
int good();
int clear(int flag=0);
```

为提高程序的可靠性，应在程序中检测 I/O 流的操作是否正常。例如用 fstream 默认方式打开文件时，如果文件不存在，fail()就能检测到错误发生，然后通过 rdstate 方法获得文件状态。

```
fstream file("test.txt");
if(file.fail())
{
	cout << file.rdstate << endl;
}
```

16.4.2 向文件追加写入

在写入文件时，有时用户不会一次性写入全部数据，而是在写入一部分数据后再根据条件向文件中追加写入，实例 16.8 将实现这一功能。

【例 16.8】 文件追加。

实例位置：光盘\MR\Instance\16\16.8

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
	ofstream ofile("test.txt", ios::app);
	if(!ofile.fail())
	{
		cout << "start write " << endl;
		ofile << "Mary ";
		ofile << "girl ";
```



```
        ofile << "20 ";
    }
    else
        cout << "can not open";
    return 0;
}
```



程序将字符串“Mary girl 20”追加到文本文件 test.txt 中，文本文件 test.txt 中的内容没有被覆盖。如果 test.txt 文件不存在则创建该文件并写入字符串“Mary girl 20”。连续运行两次程序，结果如图 16.13 和图 16.14 所示。

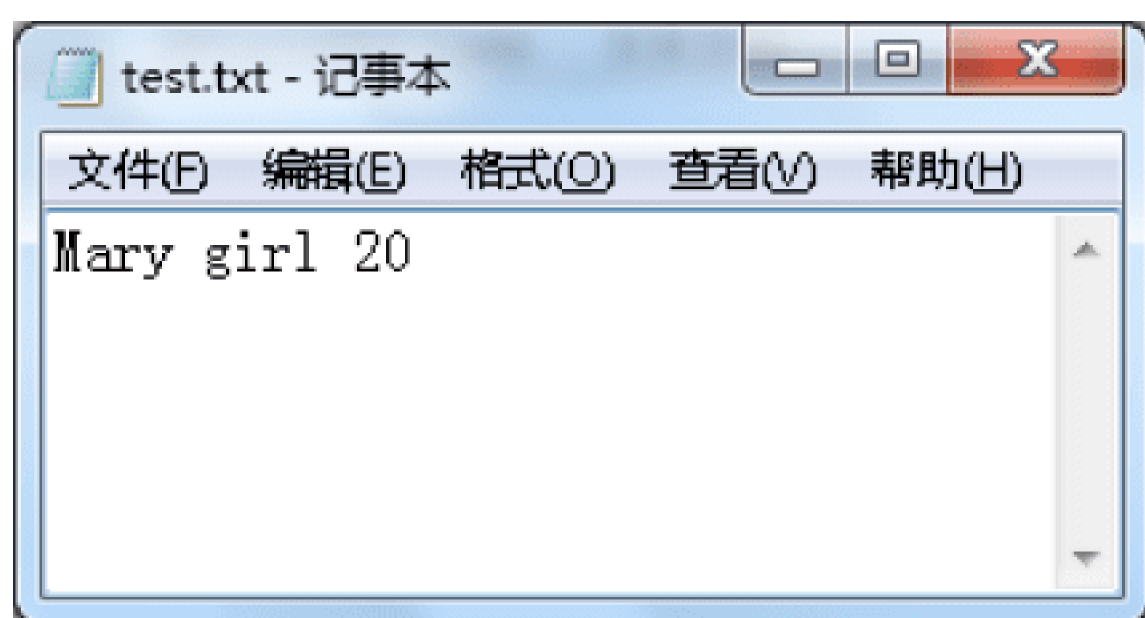


图 16.13 程序第一次执行

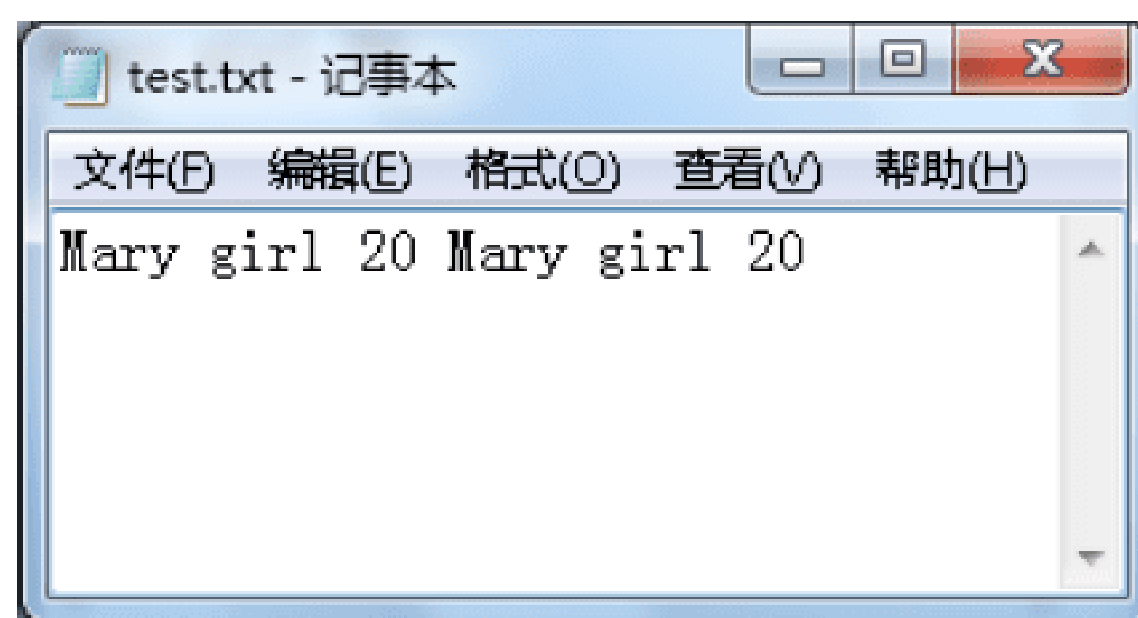


图 16.14 程序第二次执行

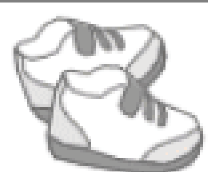
追加可以使用其他方法实现，例如先打开文件然后通过 seekp 方法将文件指针移到末尾，再向文件中写入数据，整个过程和使用参数取值一样。使用 seekp 方法实现追加的代码如下：

```
fstream iofile("test.dat",ios::in| ios::out| ios::binary);
if(iofile)
{
    iofile.seekp(0,ios::end);           //为了写入移动
    iofile << endl;
    iofile << "我是新加入的"
    iofile.seekg(0);                    //为了读取移动
    int i=0;
    char data[100];
    while(!iofile.eof && i< sizeof(data))
        iofile.get(data[i++]);
    cout << data;
}
```

程序打开 test.dat 文件，查找文件的末尾，在末尾加入字符串，然后再将文件指针移到文件开始处，输出文件的内容。

16.4.3 文件结尾的判断

在操作文件时，经常需要判断文件是否结束，使用 eof()可以实现。另外也可以通过其他方法来判断，例如使用流的 get()方法，如果文件指针指向文件末尾，get()方法获取不到数据就返回-1，这也可以作为判断结束的方法。例如：






```
ifstream ifile("test.txt");
if(!ifile)
{
	cerr << "open fail" << endl;
}
char ch;
while(ifile.get(ch))
{
	cout << ch;
}
cout << endl;
ifile.close();
```

程序实现输出 test.txt 文件的内容，同样的功能使用 eof()方法也可以实现。例如：

```
ifstream ifile("test.txt");
if(!ifile.fail())
{
	while(!ifile.eof())
	{
		char ch;
		ifile.get(ch);
		if(!ifile.eof())				//差一个空格
			cout << ch;
	}
	ifile.close();
}
```

程序仍然是输出 test.txt 文件中的内容，但使用 eof()方法需要多判断一步。

很多地方需要使用 eof()方法来判断文件是否已经读取到末尾，下面通过实例来讲述如何使用 eof()方法判断文件是否结束。

【例 16.9】 记录并输出空格位置。

实例位置：光盘\MR\Instance\16\16.9

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
	ifstream ifile("test.txt");
	if(!ifile.fail())
	{
		while(!ifile.eof())
		{
			char ch;
			streampos sp = ifile.tellg();
			ifile.get(ch);
			if(ch == ' ' )
```





```
                {
                        cout << "postion:" << sp ;
                        cout <<"is blank "<< endl;
                }
            }
        }
        return 0;
}
```

程序打开文本文件 test.txt，文件的内容如图 16.15 所示。

程序运行结果如图 16.16 所示。

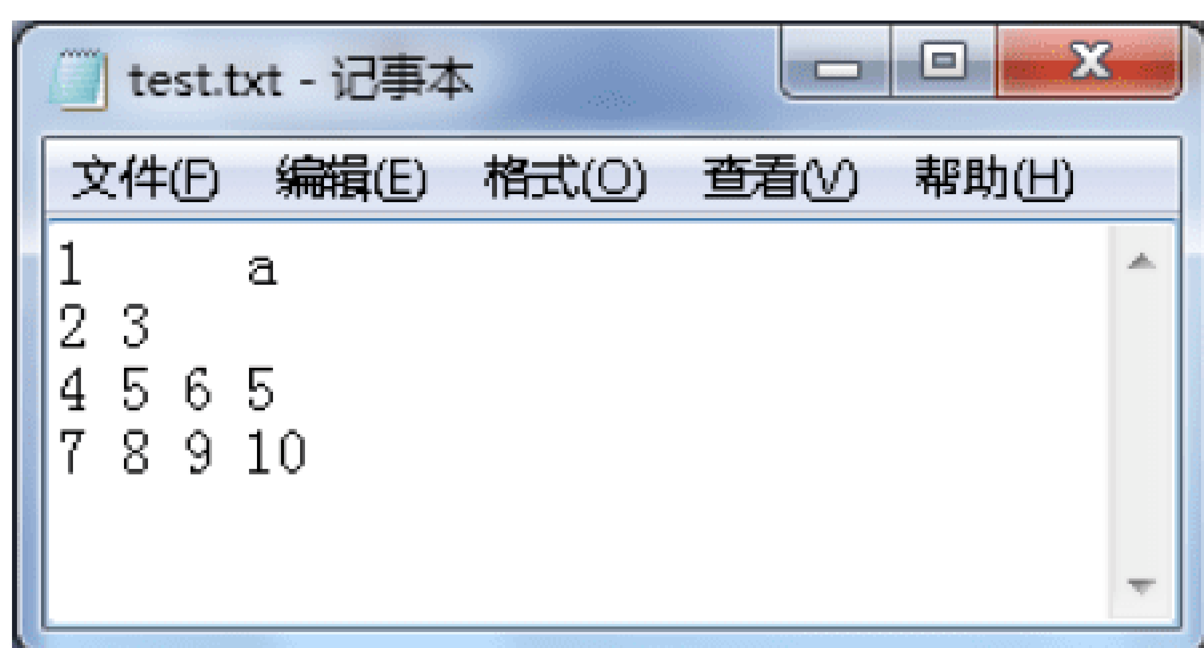


图 16.15　文本文件内容

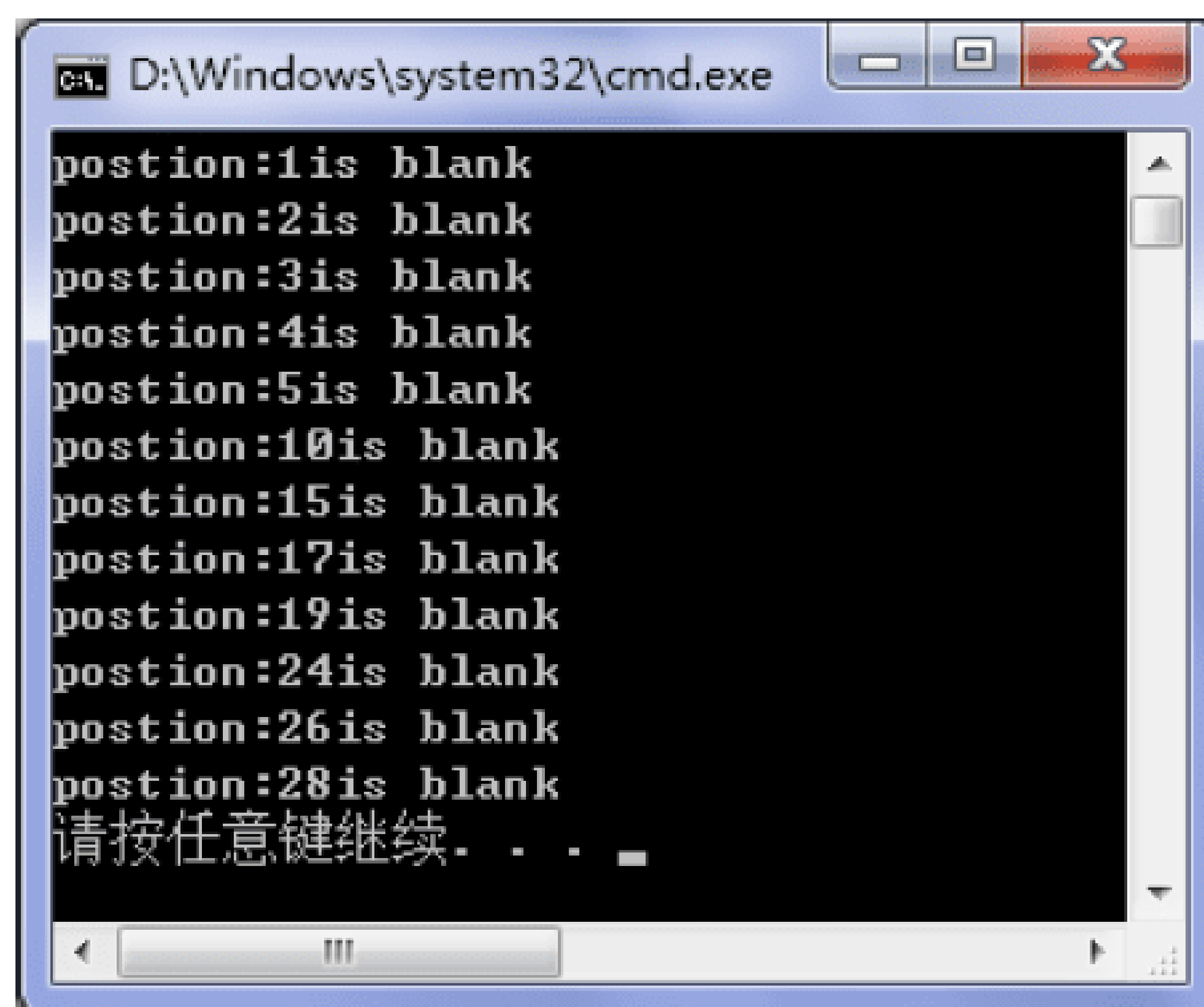


图 16.16　记录并输出空格位置

16.4.4　在指定位置读写文件

要实现在指定位置读写文件的功能，首先要了解文件指针是如何移动的，下面将介绍用于设置文件指针位置的函数。

- ☑ seekg：位移字节数，相对位置用于输入文件中指针的移动。
- ☑ seekp：位移字节数，相对位置用于输出文件中指针的移动。
- ☑ tellg：用于查找输入文件中的文件指针位置。
- ☑ tellp：用于查找输出文件中的文件指针位置。

位移字节数是移动指针的位移量，相对位置是参照位置。取值如下。

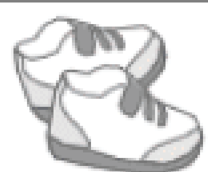
- ☑ ios::beg：文件头部。
- ☑ ios::end：文件尾部。
- ☑ ios::cur：文件指针的当前位置。

例如 seekg(0,ios::beg)是将文件指针移动到相对于文件头 0 个偏移量的位置，即指针在文件头。

【例 16.10】 输出文件指定位置的内容。

实例位置：光盘\MR\Instance\16\16.10

```
#include "stdafx.h"
#include <iostream>
```





```
#include <fstream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
        ifstream ifile;                         //打开文件的流对象
        char cFileSelect[20],ch;
        cout << "input filename:\n";
        cin >> cFileSelect;
        ifile.open(cFileSelect);                //读的方式打开文件
        if(!ifile)                              //出错判断
        {
                cout << cFileSelect << "can not open" << endl;
                return 0;
        }
        ifile.seekg(0,ios::end);                //指针调到末尾
        int maxpos = ifile.tellg();             //返回文件大小
        cout << "file length is " << maxpos << endl;
        int pos;
        cout << "Position:\n";
        cin >> pos;                             //指定要输出的位置
        if(pos > maxpos)                        //超出文件长度会报错
        {
                cout << "is over file length" << endl;
                ifile.close();                  //关闭文件
                return 0;                       //退出程序
        }
        else
        {
                ifile.seekg(pos);               //设置文件指针位置
                ifile.get(ch);                  //读取指针指向位置的字符
                cout << ch <<endl;              //输出
        }
        ifile.close();                          //关闭文件
        return 1;
}
```

当前路径下有 test.txt 文件，该文件中含有字符串“www.mingrisoft.com”，如图 16.17 所示。运行程序，输入文件名 test.txt，显示结果如图 16.18 所示。

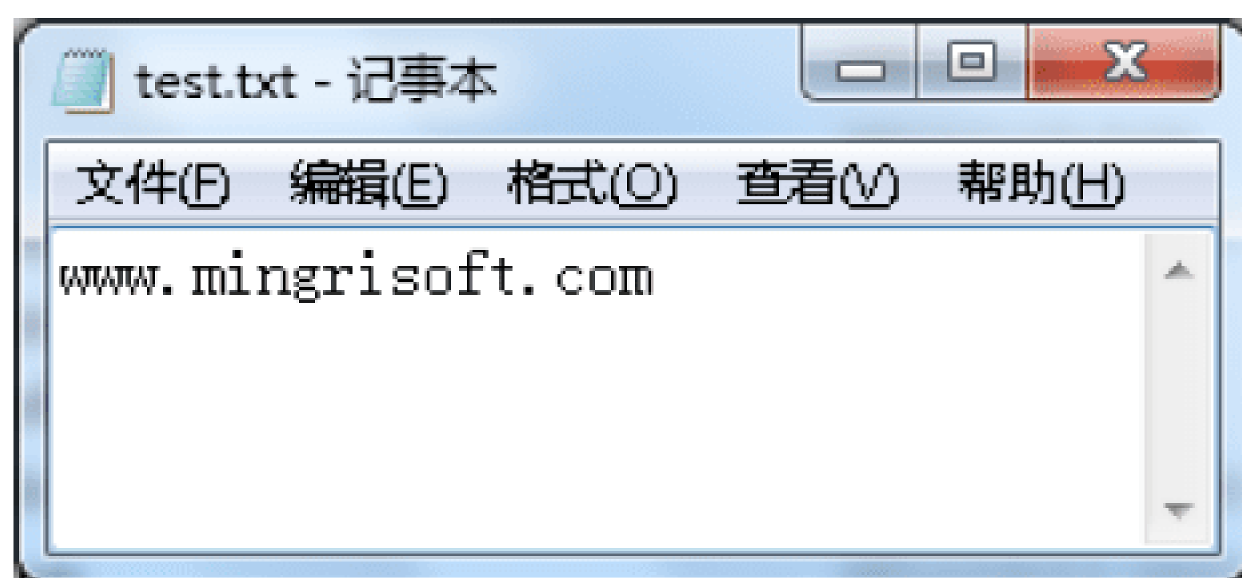


图 16.17　test.txt 文件的内容

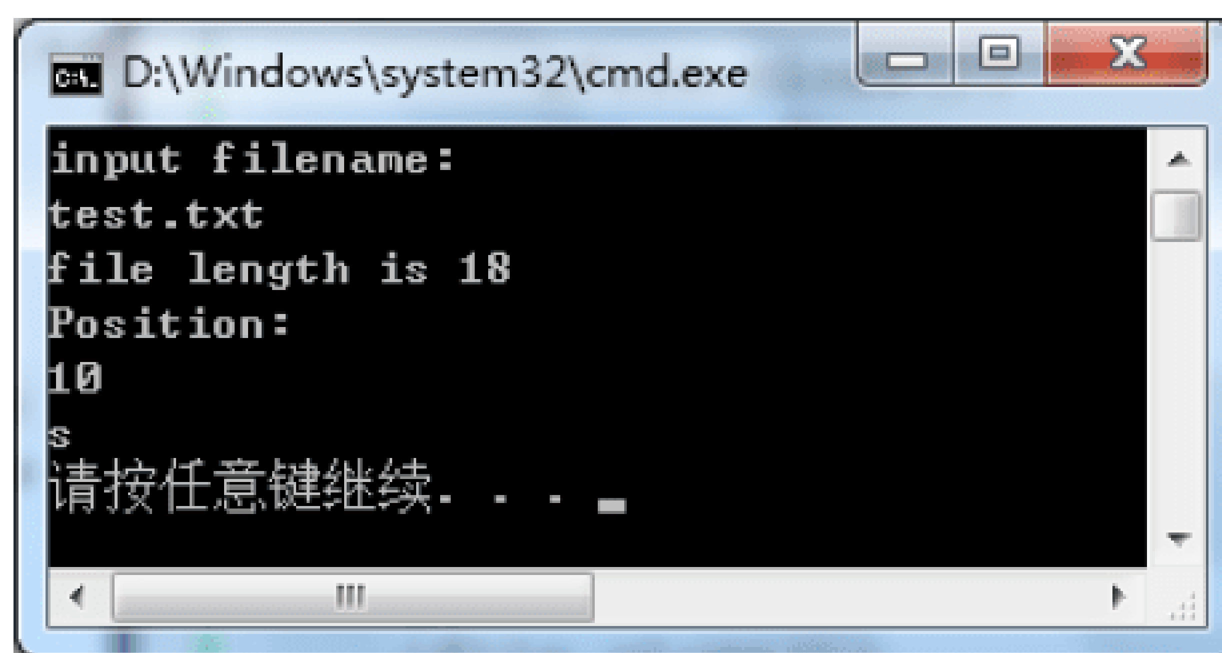
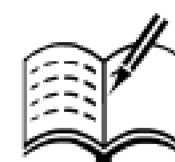


图 16.18　输出文件指定位置的内容

通过 seekg 和 tellg 可以获得文件长度，实例 16.10 用 maxpos 保存获得的文件长度，并输出



文件指针指定位置处的字符。



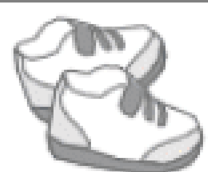
16.5　文件和流的关联和分离

一个流对象可以在不同时间表示不同文件。在构造一个流对象时，不用将流和文件绑定。使用流对象的 open 成员函数动态与文件关联，如果要关联其他文件就调用 close 成员函数关闭当前文件与流的连接，再通过 open 成员函数建立与其他文件的连接。下面通过实例来实现文件和流的关联和分离功能。

【例 16.11】 文件和流的关联和分离。

实例位置：光盘\MR\Instance\16\16.11

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
	const char* filename="test.txt";
	fstream iofile;
	iofile.open(filename,ios::in);
	if(iofile.fail())
	{
		iofile.clear();
		iofile.open(filename, ios::in| ios::out| ios::trunc);
	}
	else
	{
		iofile.close();
		iofile.open(filename, ios::in| ios::out| ios::ate);
	}
	if(!iofile.fail())
	{
		iofile << "+我是新加入的";
		iofile.seekg(0);
		while(!iofile.eof())
		{
			char ch;
			iofile.get(ch);
			if(!iofile.eof())
				cout << ch;
		}
		cout << endl;
	}
	return 0;
}
```





程序打开文本文件 test.txt，文件的内容如图 16.19 所示。

程序运行结果如图 16.20 所示。

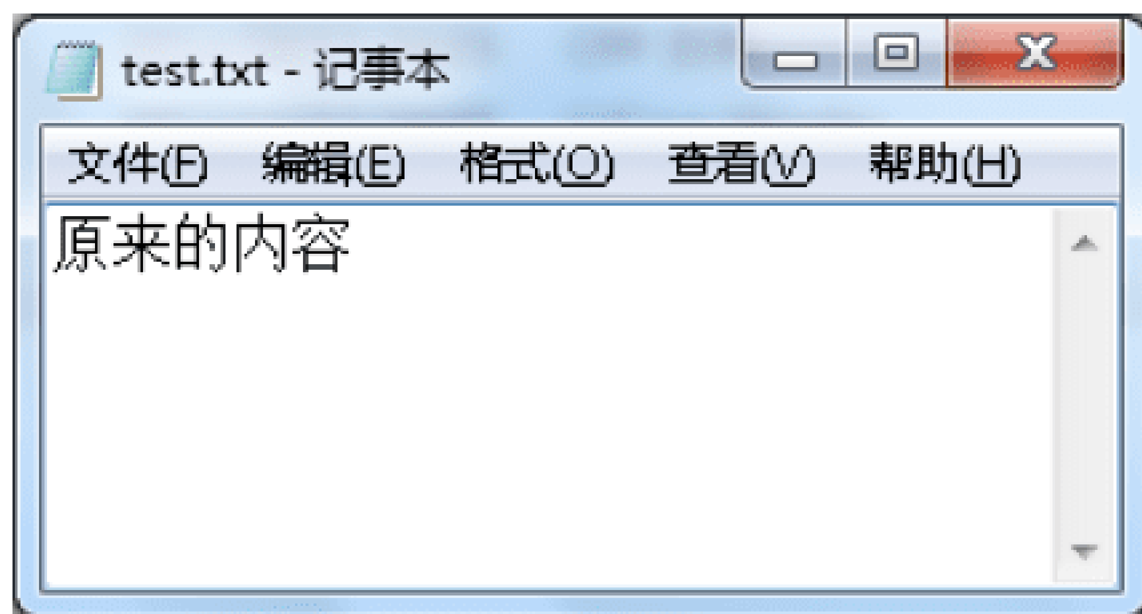


图 16.19　文件原来的内容

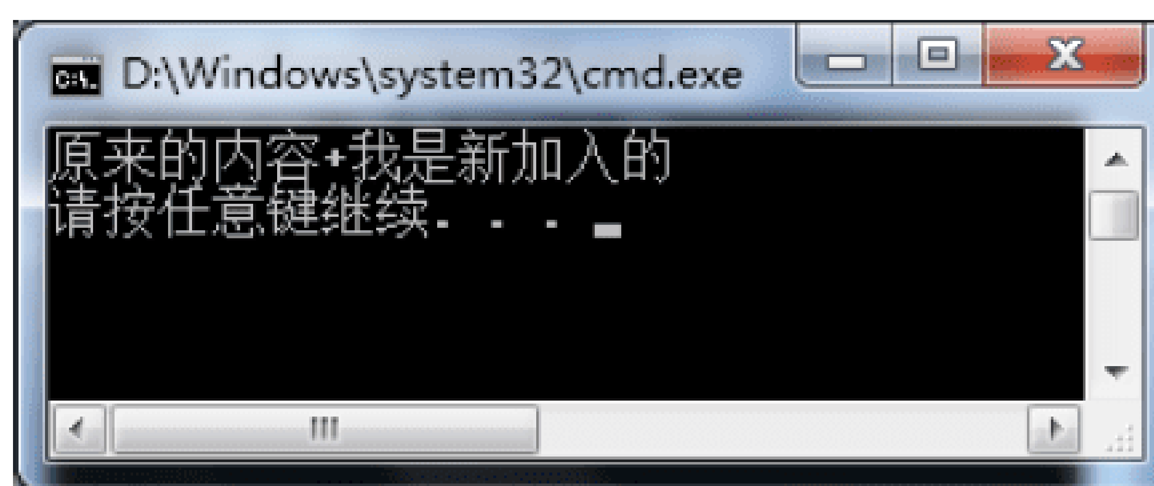


图 16.20　运行结果

程序需要用户输入文件名，然后使用 fstream 的 open()方法打开文件，如果文件不存在就通过在 open()方法中指定 ios::in| ios::out| ios::trunc 参数取值创建该文件，然后向文件中写入数据，接着将文件指针指向开始处，最后输出文件内容。程序在第一次调用 open()方法打开文件后，如果文件存在，则调用 close()方法将文件流与文件分离，接着再调用 open()方法建立文件流与文件的关联。

16.6　删除文件

前文介绍了文件的创建以及文件的读写，本节通过一个具体实例来讲述如何在程序中将一个文件删除。代码如下：

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
	char file[50];
	cout <<"Input file name："<<"\n";
	cin >>file;
	if(!remove(file))
	{
		cout <<"The file:"<<file<<"已删除"<<"\n";
	}
	else
	{
		cout <<"The file:"<<file<<"删除失败"<<"\n";
	}
}
```

程序通过 remove 函数将用户输入的文件删除。remove 函数是系统提供的函数，可以删除指定的磁盘文件。





16.7　综合应用

16.7.1　记录类的信息

【例 16.12】 可以设计一个轿车类，它具有颜色、牌照、品牌等属性。打开文件，使用文件输出流，将成员变量的值输入到文件中，记录若干个汽车的属性。关键代码如下：

实例位置：光盘\MR\Instance\16\16.12

```
class car
{
public:
	string color;
	string type;
	string ID;
	car(string color,string type,string ID)
	{
		this->color = color;
		this->type = type;
		this->ID = ID;
	}
};
```

将轿车类信息储存到文件中：

```
list<car> carList;
carList.push_back(car("红色","红旗","京 AXXXXXX"));
carList.push_back(car("蓝色","奥迪","沪 cXXXXXX"));
carList.push_back(car("绿色","宝马","吉 AXXXXXX"));

fstream file;
file.open("car.txt");
for(auto i = carList.begin();i!= carList.end();i++)
{
 file<<"颜色:"<<i->color
	<<"\t 车型:"<<i->type
	<<"\t 牌照:"<<i->ID
	<<"\n";
}
```

程序运行结果如图 16.21 所示。





Note

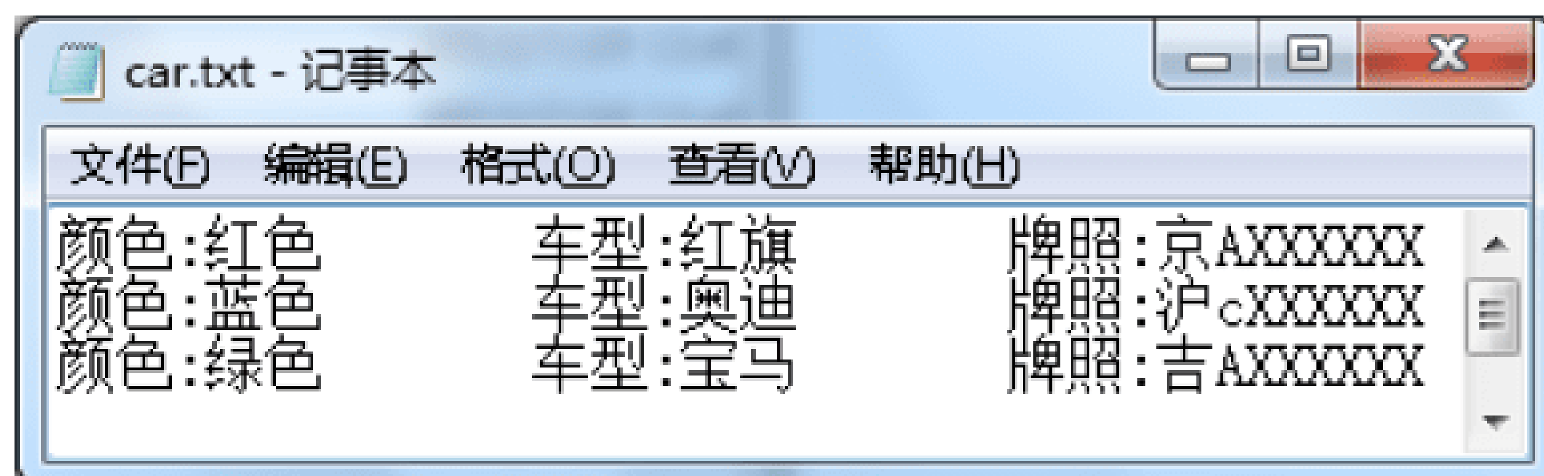


图 16.21　记录类的信息

16.7.2　读取文件信息

【例 16.13】 本例将实现对文件信息的读取并在终端上显示。在工程目录下建立一个空的 txt 文件。输入 5 个字符串，字符串间用空格分开。由于文件中字符串的格式已经确定，在程序中检查接收字符是否为空格。当接收字符为空格时，应当用另外一个字符串变量储存。关键代码如下：

实例位置：光盘\MR\Instance\16\16.13

```
string dArray[5]={};
int index = 0;
ifstream file("test.txt");      //此文件已经在文件夹下建立，可直接调用
if(!file.fail())
{
    while(!file.eof()&&index<5)
    {
        char tmp = ' ';
        file.get(tmp);
        if(tmp!=' ')
        {
            dArray[index]+=tmp;
        }
        else
        {
            cout<<dArray[index]<<endl;
            index++;
        }
    }
}
else
{
    printf("文件创建失败");
}
return 0;
```

程序运行结果如图 16.22 所示。







图 16.22　读取文件信息

16.8　本章常见错误

16.8.1　文件打开要记得关闭

文件作为系统中的一种资源，使用后要关闭，断开文件与流之间的联系，禁止再对该文件操作。打开文件会一直占用此文件资源，如果操作后不关闭，其他进程就无法对该文件操作，文件被调入内存中的数据也可能会丢失。

16.8.2　peek 不能用于 ofstream

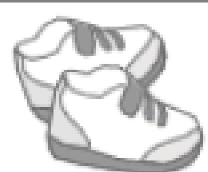
peek 用于判断文件中的下一个字符是什么，可用于文件流类 ifstream 和 fstream，但不用于 ofstream 类。

16.8.3　忘记调回指针，读不到内容

测量一个文件的大小可以用 seekg()将指针调到文件末尾，然后用 tellg()显示此时指针的位置，即文件大小。此时指针还在文件末尾，如果没有重新调整指针位置就用 get()去读取文件内容，就会什么都读不到。

16.9　本 章 小 结

本章主要介绍使用文件流进行文件操作，文件在打开时可以控制文件是为写打开还是为读打开，控制打开模式可以控制执行效率，掌握文件的随机读取就可以快速读取想要的数据，可以实现文件中数据的修改及插入。另外，本章还介绍了使用一个文件流打开多个文件的实现方法，使读者掌握文件流和文件的区别。





Note

16.10　跟 我 上 机

参考答案：光盘\MR\跟我上机

设计函数，实现对文件内容加密和解密。本例对当前路径下的文件 test.txt 做加密处理。按 1 加密，设置整型数值的密码，完成加密。按 2 解密，输入密码完成解密。到当前路径下打开 test.txt 文件以查看加密情况。

```
#include <iostream>
#include <fstream>
#define FileName "test.txt"                          //要操作的文件名
using std::cout;
using std::cin;
using std::endl;
using std::ifstream;
using std::ofstream;
using std::fstream;
void create_file(ofstream &ofile,char *fName)        //创建文件
{
	ofile.open(fName);
	if(ofile.fail())                                 //出错判断
	{
		cout<<"open failed!"<<endl;                  //如果创建失败提示信息并退出程序
		exit(0);
	}
}
void open_file(ifstream &ifile,char *fName)          //打开已存在的文件
{
	ifile.open(fName);
	if(ifile.fail())                                 //出错判断
	{
		cout<<"open failed!"<<endl;
		exit(0);
	}
}
//加密或解密
void operate(ifstream &ifile,ofstream &ofile,int PassWord,bool flag)
{
	open_file(ifile,FileName);                       //打开
	char str[100];                                   //此数组视文件大小而定
	int i = 0;
	while(ifile.peek() != EOF)                       //循环读取
	{
		char ch;
		ifile.get(ch);
		if(flag)
```






```
                ch += PassWord;                     //加密
            else
                ch -= PassWord;                     //解密
            str[i++] = ch;                          //处理后的字符存入数组
        }
        str[i] = '\0';                              //字符串最后加结束符
        ifile.close();                              //关闭
        create_file(ofile,FileName);
        ofile << str;                               //向文件里写处理后的文件内容
        ofile.close();
        if(flag)
            cout<<"加密成功\n";
        else
            cout<<"解密成功\n";
}
int main()
{
        ofstream ofile;                             //定义文件输出流对象
        ifstream ifile;                             //定义文件输入流对象
        int PassWord = 0;                           //密码
        bool flag;                                  //加密解密标志位
        cout<<"加密按 1，解密按 0\n";
        cin>>flag;                                  //选择加密、解密
        cout<<"输入密码:\n";
        cin>>PassWord;                              //输入密码
        operate(ifile,ofile,PassWord,flag);         //加密、解密
        return 0;
}
```

第 3 篇



实战篇

- 第 17 章　C++语言游戏开发
- 第 18 章　人事考勤管理系统（Visual Studio 2010 和 SQL Server 2008 实现）

第17章

C++语言游戏开发

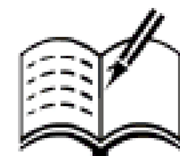
（视频讲解：2 小时 48 分钟）

本章致力于通过设计模拟 ATM 机界面、猜数字游戏和吃豆子游戏这 3 个例子使读者巩固前面所学的基础编程，灵活运用常量、变量、运算符、表达式以及几种语句，将基础知识与实际应用结合起来，提高自身的编程能力。

本章能够完成的主要范例（已掌握的在方框中打勾）

- [ ] 了解程序开发的思想
- [ ] 掌握开发 ATM 机的设计思路
- [ ] 掌握开发游戏的设计思路
- [ ] 掌握如何建立一个 Windows 窗口应用程序
- [ ] 了解函数模板和动态分配的实际用途





17.1　模拟 ATM 机界面程序

17.1.1　概述

设计一个简单的模拟自动提款机 ATM 程序设计界面，实现用户登录及取款等功能。

17.1.2　需求分析

（1）设计一个模拟自动取款机 ATM，有常用的功能。

（2）主要功能有用户输入密码登录主界面、取款功能、取款后显示取款金额以及剩余金额、退出功能等。

（3）程序执行的命令包括：

① 输入正确密码进入主目录界面。

② 执行取款界面。

③ 显示取款金额以及剩余金额界面。

④ 退出系统界面。

17.1.3　设计思路

设计一个常用的自动取款机，要包括常见的功能：取款功能、显示取款金额及剩余金额功能等。先要输入密码，密码不正确并输入次数小于 3 次将提示重新输入，否则退出程序；密码正确将进入取款界面，执行取款功能，然后显示取款金额及剩余金额，退出程序。

17.1.4　详细设计

程序主要关系图如图 17.1 所示。

程序设计代码如下：

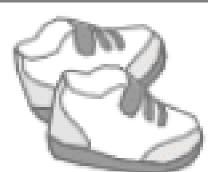
（1）创建一个 C 文件。

（2）引用头文件。

```
#include<stdio.h>
#include<stdlib.h>
```

（3）变量类型声明，分别定义了字符型和基本整型变量。

```
char Key,CMoney;
    int password,password1=123,i=1,a=1000;                          /*定义变量*/
```





Note

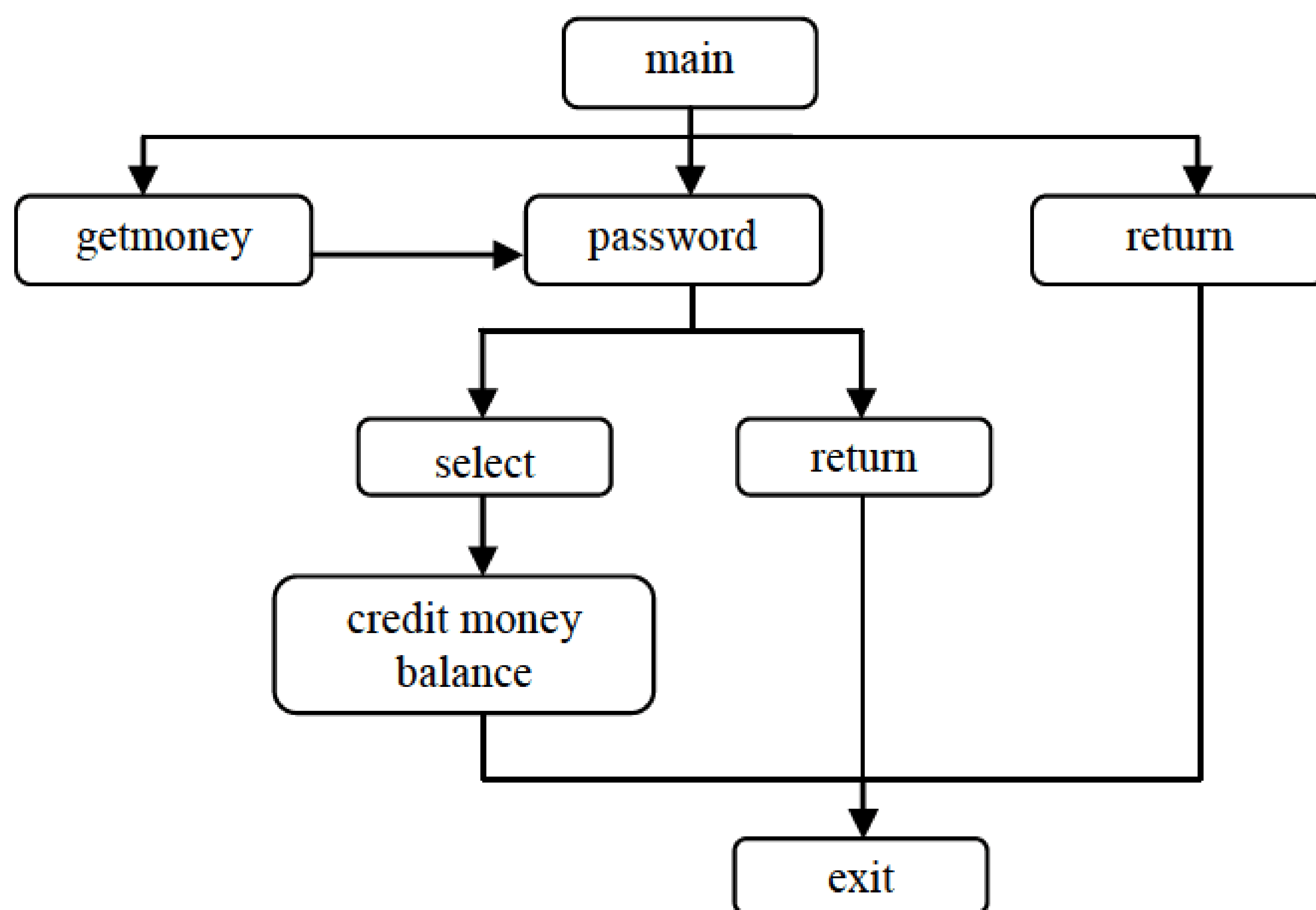


图 17.1　程序主要关系图

（4）使用 do…while 循环，当输入数据不是 1、2、3 中任意一个时，将始终进行 do 循环体中的语句，否则进行下面 switch 语句的操作。

```
do{
        system("cls");
        printf("*************************\n");
        printf("*   Please select key:    *\n");
        printf("*   1. password            *\n");
        printf("*   2. get money           *\n");
        printf("*   3. Return              *\n");
        printf("*************************\n");
        Key = getchar();
   }while( Key!='1' && Key!='2' && Key!='3' );
      /*当输入值不是 1、2、3 中任意值时显示 do 循环体中的内容*/
```

程序实现功能如图 17.2 所示。

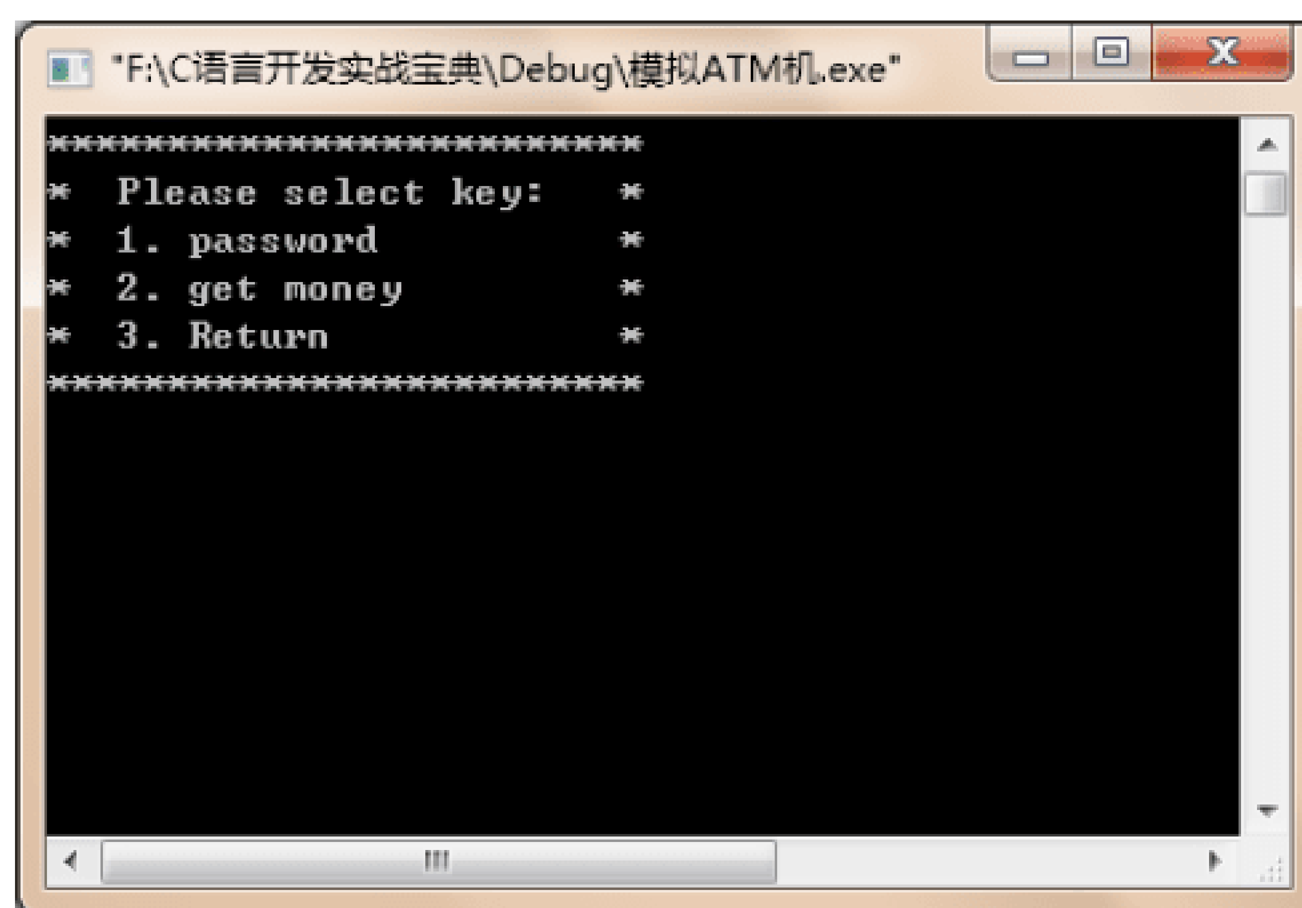
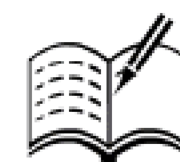


图 17.2　系统登录

（5）使用 switch 语句构成本程序的选择功能，当输入 1 时进行用户密码确认，此时用到 if 语句判断输入密码是否正确及输入密码次数是否超过 3 次。





Note

```
switch(Key)
        {
        case '1':                                       /*输入值为 1 时执行 case1*/
            system("cls");
            do
            {
                i++;
                printf("      please input password      ");
                scanf("%d",&password);
                if(password1!=password)                 /*如果输入密码不正确，执行下面语句*/
                {
                    if(i>3)                             /*如果 3 次密码输入均不正确将退出程序*/
                    {
                        printf(" Wrong! Press any key to exit...   ");
                        getchar();
                        exit(0);
                    }
                    else
                        puts("wrong,try again");        /*输入次数未到 3 次，可继续输入*/
                }
            }
            while(password1!=password&&i<=3);
            /*如果密码不正确且输入次数小于等于 3 次，执行 do 循环体中语句*/
            printf("OK! Press any key to continue…   "); /*密码正确返回初始界面开始其他操作*/
            getchar();
```

程序实现功能如图 17.3 所示。

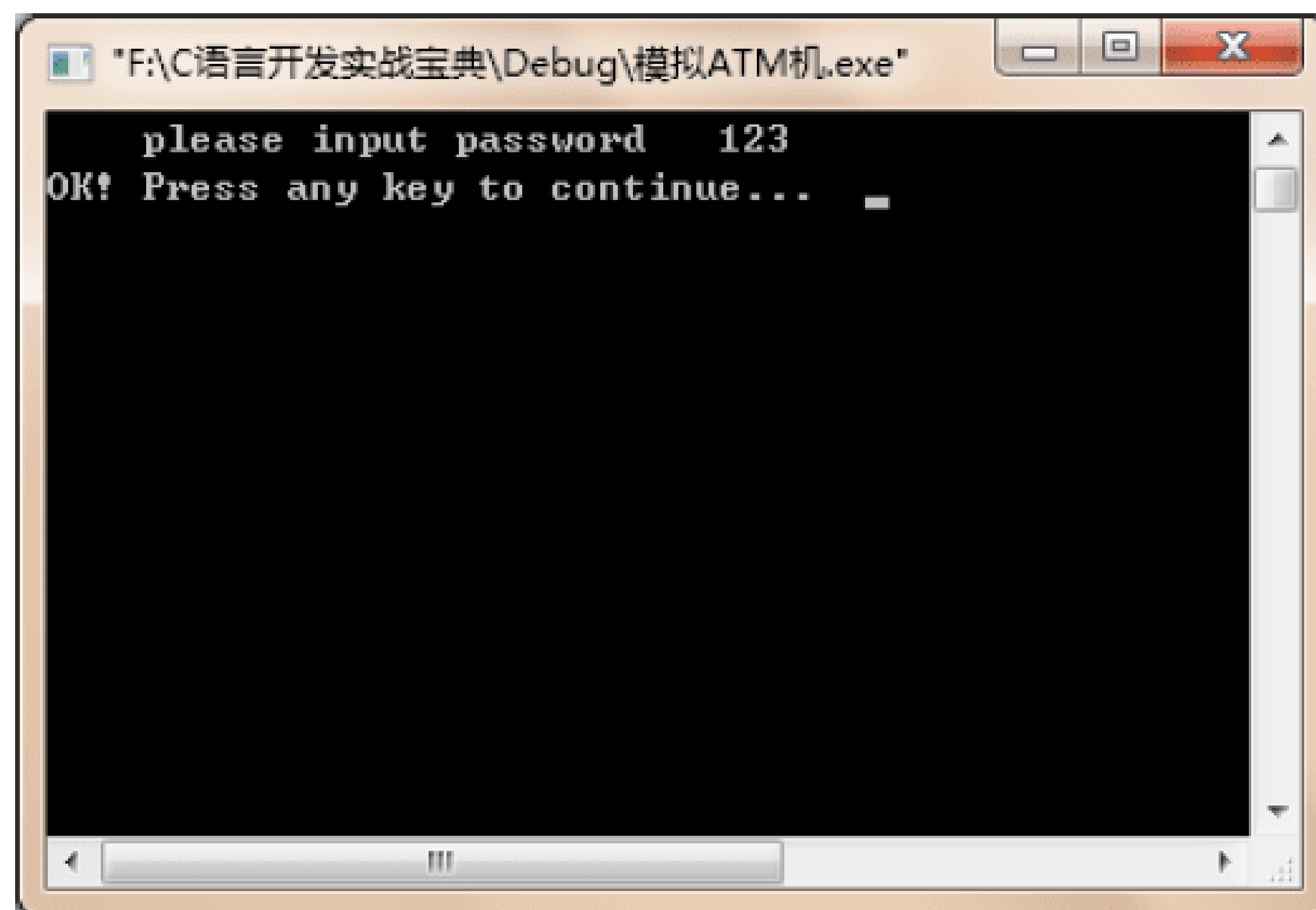
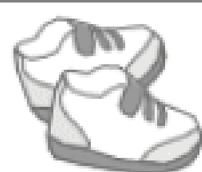


图 17.3　判断密码是否正确

（6）当输入 2 时进行用户取款操作，此时再次使用 do…while 循环和 switch 构成取款选择界面，根据取款金额输入不同数据，输入数据 4 时，第一个 break 跳出里层 switch 循环，第二个 break 跳出外层 switch 循环，回到最初 while 控制下的主界面。

```
case '2':                                               /*输入值为 2 时执行 case2*/
            do{
                system("cls");
                if(password1!=password)
```






```
        /*如果在 case1 中密码输入不正确将无法进行后面操作*/
    {printf("please logging in,press any key to continue...");
    getchar();
    break;}
    else
    {
        printf("*********************************\n");
        printf("     Please select:                  *\n");
        printf("*      1. $100                        *\n");
        printf("*      2. $200                        *\n");
        printf("*      3. $300                        *\n");
        printf("*      4. Return                      *\n");
        printf("*********************************\n");
        CMoney = getchar();                          }
}while( CMoney!='1' && CMoney!='2' && CMoney!='3'&&CMoney!='4');
/*输入值不是 1、2、3、4 中任意数将继续执行 do 循环体中语句*/
switch(CMoney)
{
case '1':                                        /*输入 1 时执行 case1 中的操作*/
    system("cls");
    a=a-100;
    printf("******************************************\n");
    printf("*   Your Credit money is $100,Thank you!   *\n");
    printf("*                The balance is $%d.            *\n",a);
    printf("*              Press any key to return...       *\n");
    printf("******************************************\n");
    getchar();
    break;
case '2':
    system("cls");                               /*输入 2 时执行 case2 中的操作*/
    a=a-200;
    printf("******************************************\n");
    printf("*   Your Credit money is $200,Thank you!   *\n");
    printf("*                  The balance is $%d.          *\n",a);
    printf("*              Press any key to return...       *\n");
    printf("******************************************\n");
    getchar();
    break;
case '3':                                        /*输入 3 时执行 case3 中的操作*/
    system("cls");
    a=a-300;
    printf("******************************************\n");
    printf("*   Your Credit money is $300,Thank you!      *\n");
    printf("*                    the balance is $%d          *\n",a);
    printf("*                Press any key to return...     *\n");
    printf("******************************************\n");
    getchar();
    break;
case '4':                                        /*输入 4 时执行 case4 中的操作*/
    break;
```





```
            }
            break;
```

程序实现功能如图 17.4 所示。

取款界面如图 17.5 所示。

取款金额及剩余金额界面如图 17.6 所示。

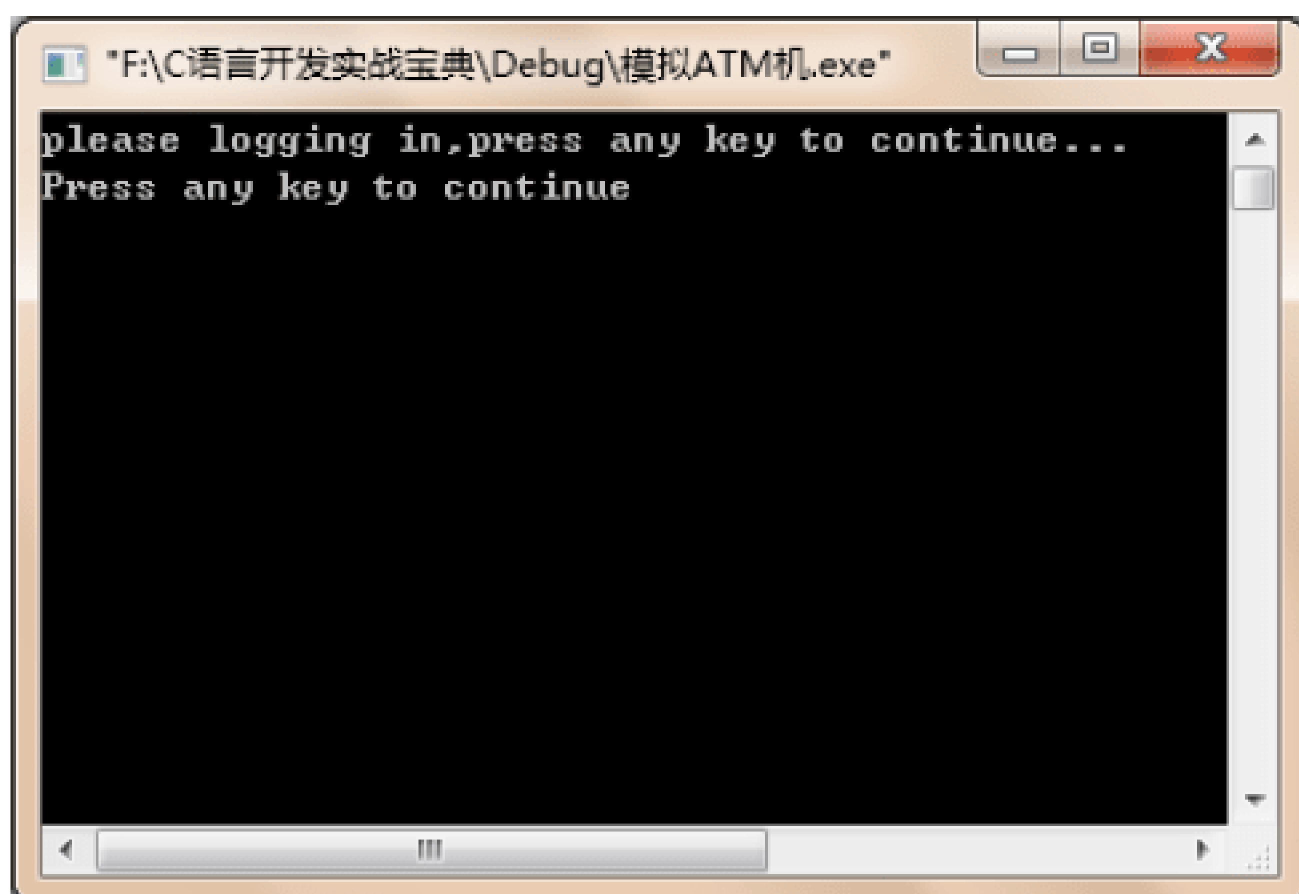


图 17.4　输入密码进行后面操作



图 17.5　取款界面

（7）当输入 3 时，第一个 break 跳出 switch 循环，第二个 break 跳出 while 循环，结束本程序。

```
case '3':
            printf("**********************************************\n");
            printf("*      Thank you for your using!              *\n");
            printf("*                  Goodbye!                   *\n");
            printf("**********************************************\n");
            getchar();
            break;
```

程序实现功能如图 17.7 所示。



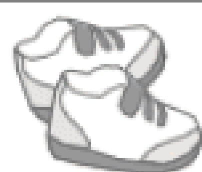
图 17.6　取款金额及剩余金额

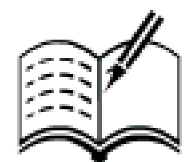


图 17.7　退出界面

17.1.5　程序代码

完整程序代码如下：





Note

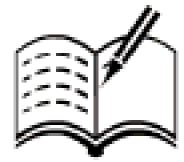
实例位置：光盘\MR\Instance\17\17.1

```
#include<stdio.h>
#include<stdlib.h>
main()
{
    char Key,CMoney;
    int password,password1=123,i=1,a=1000;                  /*定义变量*/
    while(1)
    {
        do{
            system("cls");
            printf("*************************\n");
            printf("*    Please select key:      *\n");
            printf("*    1. password              *\n");
            printf("*    2. get money             *\n");
            printf("*    3. Return                *\n");
            printf("*************************\n");
            Key = getchar();
        }while( Key!='1' && Key!='2' && Key!='3' );
        /*当输入值不是 1、2、3 中任意值时显示 do 循环体中的内容*/
        switch(Key)
        {
        case '1':                                           /*输入值为 1 时执行 case1*/
            system("cls");
            do
            {
                i++;
                printf("      please input password      ");
                scanf("%d",&password);
                if(password1!=password)                     /*如果输入密码不正确，执行下面语句*/
                {
                    if(i>3)                                 /*如果 3 次密码输入均不正确将退出程序*/
                    {
                        printf(" Wrong! Press any key to exit...    ");
                        getchar();
                        exit(0);
                    }
                    else
                        puts("wrong,try again");            /*输入次数未到 3 次，可继续输入*/
                }
            }
            while(password1!=password&&i<=3);
            /*如果密码不正确且输入次数小于等于 3 次，执行 do 循环体中语句*/
            printf("OK! Press any key to continue…    ");   /*密码正确返回初始界面开始其他操作*/
            getchar();
        case '2':                                           /*输入值为 2 时执行 case2*/
            do{
                system("cls");
                if(password1!=password)
```





```
            /*如果在 case1 中密码输入不正确将无法进行后面操作*/
        {printf("please logging in,press any key to continue...");
        getchar();
        break;}
        else
        {
            printf("**********************************\n");
            printf("    Please select:               *\n");
            printf("*     1. $100                    *\n");
            printf("*     2. $200                    *\n");
            printf("*     3. $300                    *\n");
            printf("*     4. Return                  *\n");
            printf("**********************************\n");
            CMoney = getchar();                      }
}while( CMoney!='1' && CMoney!='2' && CMoney!='3'&&CMoney!='4');
/*但输入值不是 1、2、3、4 中任意数将继续执行 do 循环体中语句*/
switch(CMoney)
{
case '1':                                   /*输入 1 时执行 case1 中的操作*/
        system("cls");
        a=a-100;
        printf("********************************************\n");
        printf("*   Your Credit money is $100,Thank you!   *\n");
        printf("*              The balance is $%d.           *\n",a);
        printf("*             Press any key to return...   *\n");
        printf("********************************************\n");
        getchar();
        break;
case '2':
        system("cls");                      /*输入 2 时执行 case2 中的操作*/
        a=a-200;
        printf("********************************************\n");
        printf("*   Your Credit money is $200,Thank you!   *\n");
        printf("*               The balance is $%d.         *\n",a);
        printf("*             Press any key to return...   *\n");
        printf("********************************************\n");
        getchar();
        break;
case '3':                                   /*输入 3 时执行 case3 中的操作*/
        system("cls");
        a=a-300;
        printf("********************************************\n");
        printf("*   Your Credit money is $300,Thank you!     *\n");
        printf("*                the balance is $%d          *\n",a);
        printf("*              Press any key to return...    *\n");
        printf("********************************************\n");
        getchar();
        break;
case '4':                                   /*输入 4 时执行 case4 中的操作*/
        break;
```


Note

```
                }
                break;
                case '3':
                    printf("*******************************************\n");
                    printf("*     Thank you for your using!             *\n");
                    printf("*                  Goodbye!                 *\n");
                    printf("*******************************************\n");
                    getchar();
                    break;
            }
            break;
        }
}
```

17.1.6 小结

本章模拟了 ATM 机界面，不仅使读者巩固了前面所学的基础编程知识，而且通过反复运用循环、分支语句，加深了读者对流程控制语句的理解，提高了编程能力。

17.2 猜数字游戏

17.2.1 概述

猜数字（又称 Bull and Cows）是一种大概于 20 世纪中期兴起于英国的益智类小游戏。一般由两个人玩，也可以由一个人和电脑玩，在纸上、网上都可以玩。该游戏规则简单，但可以考验人的严谨性和耐心。

游戏规则为：参与游戏的两方，一方出数字，一方猜数字。出数字的人要想好一个没有重复数字的 4 位数，不能让猜的人知道。猜的人就可以开始猜，每猜一个数字，出数者就要根据这个数字给出几 A 几 B，其中 A 前面的数字表示位置正确的个数，而 B 前面的数字表示数字正确而位置不对的数的个数。

例如，正确答案为 5346，猜数字的人猜的数字为 5238，则提示 1A1B，其中有一个 5 位置对了，记为 1A，而 3 数字对了，而位置不对，因此即为 1B，合起来就是 1A1B。

接着，猜数字的人根据出题者的几 A 几 B 继续猜，直到猜中（即 4A0B）为止。

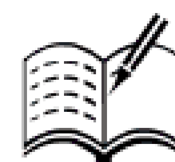
17.2.2 需求分析

通过调查，要求系统具有以下功能：

☑ 为了体现良好的娱乐性，因此要求系统具有良好的人机交互界面。



- ☑ 完全人性化设计，无须专业人士指导，即可操作本系统。
- ☑ 自动完成胜负判断，避免人为错误。



17.2.3 系统设计

1. 设计目标

本系统属于典型的小游戏，是针对单机版开发的智益小游戏。通过本系统可以达到以下目标：

- ☑ 灵活的操作，可以自动判断胜负。
- ☑ 良好的人机对话模拟，界面设计美观友好。
- ☑ 运行稳定、安全可靠。

2. 开发及运行环境

- ☑ 系统开发平台：Visual Studio 2010。
- ☑ 运行平台：Windows XP/ Win7。
- ☑ 分辨率：最佳效果 1024×768。

17.2.4 程序预览

猜数字游戏具体要求如下：开始时应输入要猜的数字的位数，这样计算机可以根据输入的位数随机地分配一个符合要求的数据，计算机输出 guess 后便可以输入数字，注意数字间需用空格或回车加以区分，计算机会根据输入信息给出相应的提示信息：A 表示位置与数字均正确的个数，B 表示位置不正确但数字正确的个数，这样便可以根据提示信息进行下次输入，直到正确为止，这时会根据输入的次数给出相应的评价。

运行程序，首先进入到程序的主界面，如图 17.8 所示。在该界面用户可以选择进入不同的菜单，选择 1，开始游戏；选择 2，查看游戏规则；选择 3，退出程序。

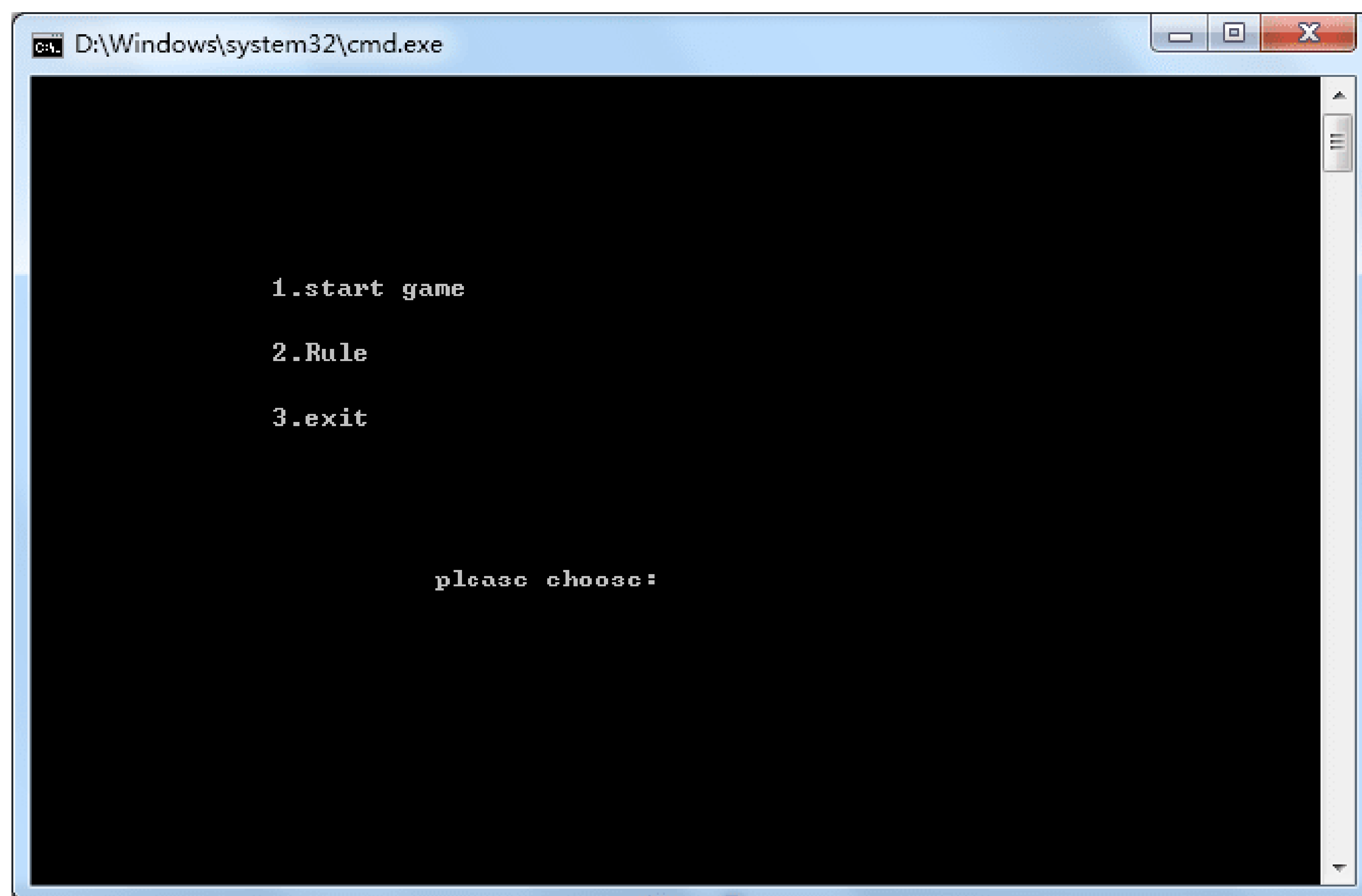


图 17.8 程序的主界面





选择 1，开始游戏。进入到游戏中，首先输入要猜的数字的个数，一般为 4 个数字，这里输入 4，当弹出 guess 提示符时，开始猜数字，并根据猜测数据提示几 A 几 B，当玩家在 5 次以内，包括 5 次猜中了数字，则提示 you are genius!（你是个天才），如图 17.9 所示。

```
D:\Windows\system32\cmd.exe
please input n:
4
guess: 1 2 3 4
clue on:0 A 4 B

guess:2 1 4 3
clue on:0 A 4 B

guess:4 3 2 1
clue on:0 A 4 B

guess:3 4 1 2
clue on:4 A 0 B

you are genius!!

按任意键继续
```

图 17.9　玩家 5 次以内猜中（依照各种算法，5 次以内都有很大的运气成分）

当玩家在 6～10 次以内猜中数字时，将提示 you are clear!（你真聪明!），效果如图 17.10 所示。

```
D:\Windows\system32\cmd.exe
4
guess:1 2 3 4
clue on:1 A 0 B

guess:5 6 7 8
clue on:0 A 3 B

guess:1 8 6 7
clue on:1 A 1 B

guess:5 2 6 7
clue on:2 A 1 B

guess:5 2 8 6
clue on:1 A 2 B

guess:8 2 7 5
clue on:2 A 2 B

guess:8 2 5 7
clue on:4 A 0 B

you are cleaver!!

按任意键继续
```

图 17.10　玩家 6～10 次猜中数字

有些数字不好猜，玩家可以超过 10 次才猜中，此时提示玩家 you need try hard!（继续努力!）。如图 17.11 所示。

当在主界面中选择 2，即可查看猜数字游戏的游戏规则，效果如图 17.12 所示。






```
D:\Windows\system32\cmd.exe

guess:7 6 0 3
clue on:0 A 2 B

guess:6 8 0 4
clue on:0 A 2 B

guess:6 5 0 8
clue on:0 A 3 B

guess:5 6 9 8
clue on:2 A 1 B

guess:6 9 0 5
clue on:0 A 4 B

guess:0 6 9 5
clue on:1 A 3 B

guess:5 0 9 6
clue on:4 A 0 B

you need try hard!!

按任意键继续
```

图 17.11　玩家超过 10 次猜中

```
D:\Windows\system32\cmd.exe
                  The Rules Of The Game
step1: input the number of digits
step2: input the number,separated by a space between two numbers
step3: A represent location and data are correct
       B represent location is correct but data is wrong!
```

图 17.12　游戏规则

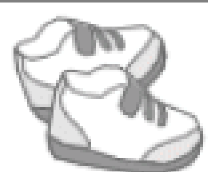
如果用户在主界面选择 3，则直接退出程序。

17.2.5　设计思路

本程序的关键是如何实现随机分配数据与核对输入数据的过程，利用系统时钟作为随机数的种子，将每次产生的 0～9 之间（包含 0 和 9）的随机数存到数组 a 中，将从键盘中输入的数字存到数组 b 中，用数组 b 中的所有数与数组 a 中的每个数比较，通过统计位置与数据均相同的个数及统计位置不同但数据相同的个数来输出提示信息。玩游戏者可以根据提示信息调整输入的数据，当输入的所有数据与所产生的随机数全部相等（位置与数据均相等）时，根据输入猜测的次数给出相应的评价，以上就是设计猜数字游戏的核心算法。

17.2.6　文件引用

在猜数字游戏中需要引用一些头文件，这些头文件可以帮助程序更好地运行。头文件的引用是通过#include 命令来实现的，下面即为本程序中所引用的头文件：



```
#include"stdafx.h"                    /*已包含输入/输出函数的头文件 stdio.h*/
#include<stdlib.h>                    /*常用子程序*/
#include<conio.h>                     /*调用 DOS 控制台 I/O*/
#include <time.h>
#include <windows.h>
```



Note

17.2.7 主要功能实现

1. 主函数

运行程序，首先进入到程序的主界面，如图 17.13 所示。在该界面用户可以选择进入不同的菜单，选择 1，开始游戏；选择 2，查看游戏规则；选择 3，退出程序。



图 17.13　程序的主界面

主程序可以用图 17.14 所示的流程图加以描述。

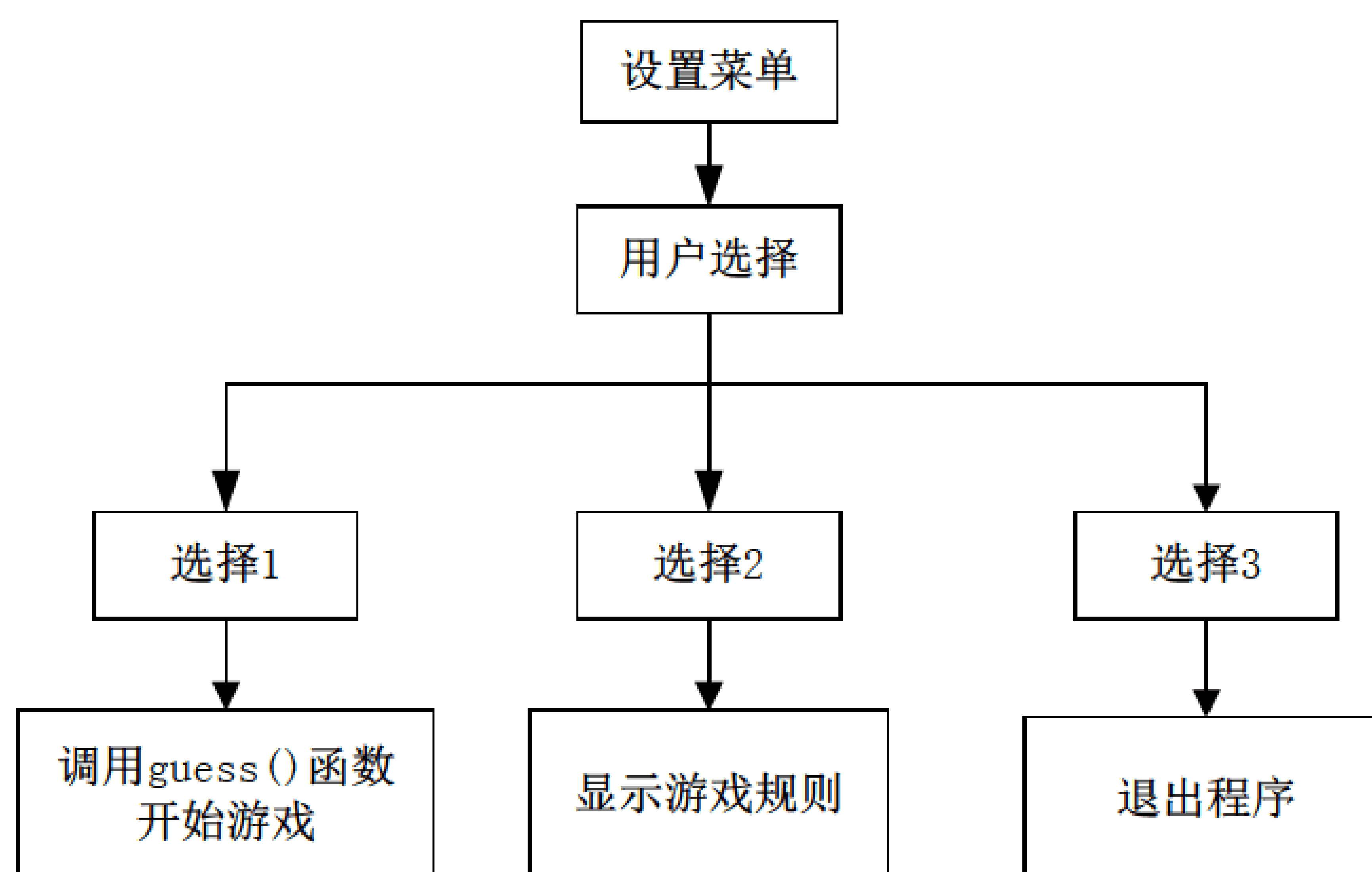


图 17.14　主程序的执行流程





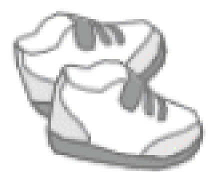
main()函数作为程序的入口函数，通过输入相应的数字选择不同的功能，程序代码如下：

```
void main()
{
    int i, n;                                          /*定义整型变量*/
    while (1)                                          /*循环*/
    {
        system("cls");                                 /*清屏*/
        gotoxy(15, 6);                                 /*将光标定位*/
        printf("1.start game");                        /*1 开始游戏*/
        gotoxy(15, 8);                                 /*光标定位*/
        printf("2.Rule");                              /*2 显示规则*/
        gotoxy(15, 10);                                /*光标定位*/
        printf("3.exit\n");                            /*3 退出*/
        gotoxy(25, 15);
        printf("please choose:");                      /*提示用户选择*/
        scanf("%d", &i);                               /*接收用户输入信息*/
        switch (i)                                     /*根据不同的输入执行不同的操作*/
        {
        case 1:                                        /*如果用户选择 1，则开始游戏*/
            system("cls");                             /*清屏*/
            printf("please input n:\n");               /*输入使用的数字个数*/
            scanf("%d", &n);                           /*接收用户输入*/
            guess(n);                                  /*调用 guess 函数*/
            Sleep(5);                                  /*程序停止 5 秒钟*/
            break;                                     /*跳出*/
        case 2:                                        /*输出游戏规则*/
            system("cls");                             /*清屏*/
            printf("\t\tThe Rules Of The Game\n");     /*显示游戏规则*/
            printf(" step1: input the number of digits\n");
            printf(" step2: input the number,separated by a space between two numbers\n");
            printf(" step3: A represent location and data are correct\n");
            printf("\tB represent location is correct but data is wrong!\n");
            Sleep(8000);                               /*暂停*/
            break;                                     /*跳出*/
        case 3:                                        /*退出游戏*/
            exit(0);                                   /*退出*/
        default:                                       /*默认*/
            break;                                     /*跳出*/
        }
    }
}
```

2．猜数字

在主界面，玩家选择 1，开始游戏。进入到游戏中，首先输入要猜的数字的个数，一般为 4 个数字，这里输入 4，当弹出 guess 提示符时，开始猜数字，并根据猜测数据提示几 A 几 B，当玩家在 5 次以内，包括 5 次猜中了数字，则提示 you are genius!（你是个天才），如图 17.15 所示。

在这个过程中将调用 guess(int n)自定义过程来执行猜数字的游戏环节。在这个游戏环节中玩家首先需要确定使用的是几位的数字，一般都是使用 4 位数字，然后程序会生成 4 位的随机数，





Note

检查这 4 个数字有没有重复的，如果有则重新选择，如果没有，则将这 4 位随机数保存到数组中。

接着提示玩家输入数字，此时玩家输入 0～9 范围内的数字，并用空格隔开。玩家每输入一次，计数变量就累加 1。并利用循环语句检测位置相同和数字相同的数字个数，并输出几 A 几 B。

当 A 的标识变量 acount 与用户输入的数字个数相同时，则根据玩家输入的次数显示不同的提示信息。

猜数字的执行过程如图 17.16 所示。

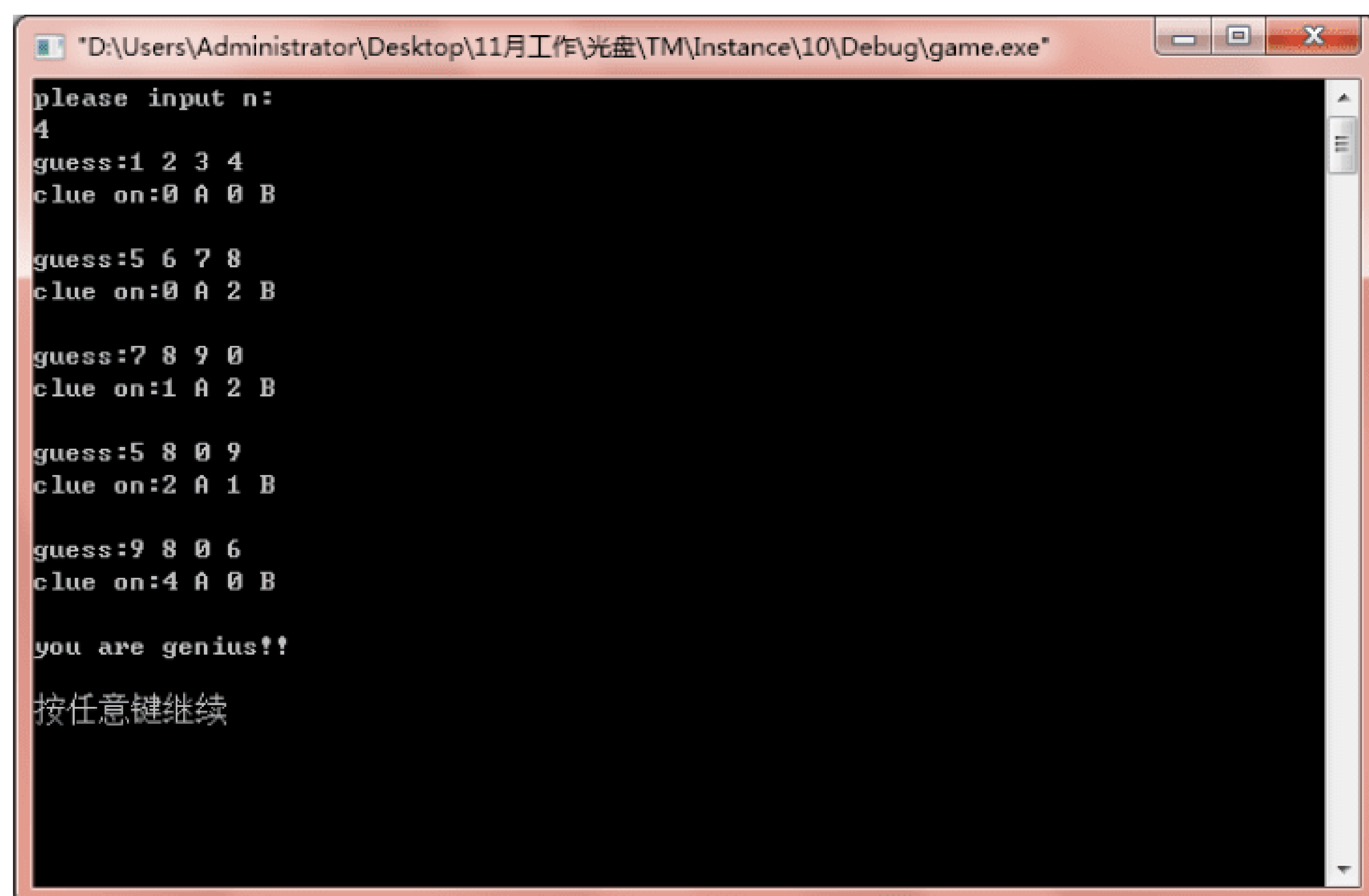


图 17.15　玩家 5 次以内猜中

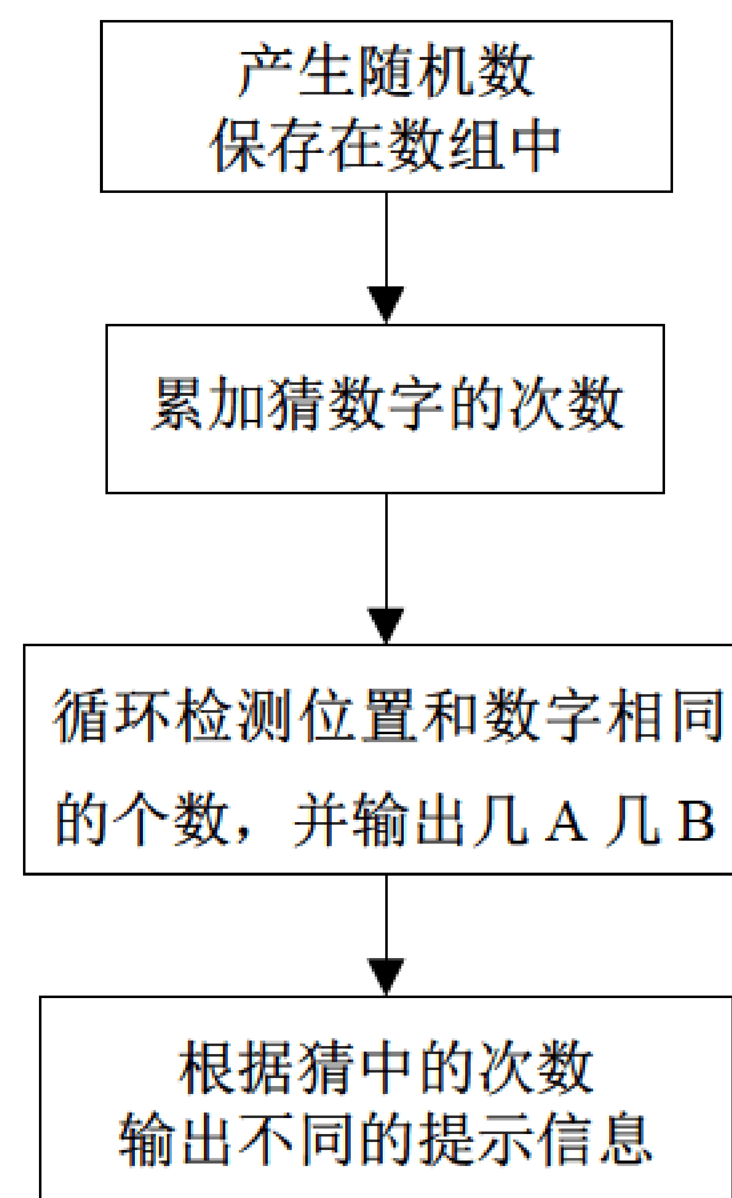
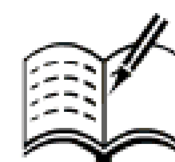


图 17.16　猜数字函数的执行过程

自定义 guess()函数，作用是产生随机数并将输入的数与产生的数作比较，并将比较后的提示信息输出。

```
void guess(int n)
{
    int acount,bcount,i,j,k=0,flag,a[10],b[10];
    do
    {
        flag=0;
        srand((unsigned)time(NULL));        /*利用系统时钟设定种子*/
        for(i=0;i<n;i++)
            a[i]=rand()%10;                 /*每次产生 0～9 范围内任意的一个随机数并存到数组 a 中*/
        for(i=0;i<n-1;i++)
        {
            for(j=i+1;j<n;j++)
                if(a[i]==a[j])              /*判断数组 a 中是否有相同数字*/
                {
                    flag=1;                 /*若有上述情况则标志位置 1*/
                    break;
                }
        }
    }while(flag==1);                        /*若标志位为 1 则重新分配数据*/
    do
```





```
    {
        k++;                                        /*记录猜数字的次数*/
        acount=0;                                   /*每次猜的过程中位置与数字均正确的个数*/
        bcount=0;                                   /*每次猜的过程中位置不正确但数字正确的个数*/
        printf("guess:");
        for(i=0;i<n;i++)
            scanf("%d",&b[i]);                      /*输入猜测的数据到数组 b 中*/
        for(i=0;i<n;i++)
            for(j=0;j<n;j++)
            {
                if(a[i]==b[i])
/*检测输入的数据与计算机分配的数据相同且位置相同的个数*/
                {
                    acount++;
                    break;
                }
                if(a[i]==b[j]&&i!=j)
/*检测输入的数据与计算机分配的数据相同但位置不同的个数*/
                {
                    bcount++;
                    break;
                }
            }
        printf("clue on:%d A %d B\n\n",acount,bcount);
        if(acount==n)                               /*判断 acount 是否与数字的个数相同*/
        {
            if(k==1)                                /*如果用户一次就输入正确*/
                printf(" you are the topmost rung of Fortune's ladder!! \n\n");
            else if(k<=5)                           /*如果用户 5 次以内猜正确*/
                printf("you are genius!!\n\n");
            else if(k<=10)                          /*如果用户 10 次以内猜正确*/
                printf("you are cleaver!!\n\n");
            else                                    /*其他情况*/
                printf("you need try hard!!\n\n");
                  getchar();                        /*接收换行字符*/
                  printf("按任意键继续");
                  getchar();                        /*等待接收指令*/
            break;
        }
    }while(1);
}
```

3．光标定位

使用 Windows API 函数封装，程序代码如下：

```
void gotoxy(int &&x,int &&y)
{
    COORD coord={x,y};
```



```
    SetConsoleCursorPosition(GetStdHandle(STD_OUTPUT_HANDLE),coord);
}
```



Note

17.2.8 小结

本节通过一个简单的小游戏将前面介绍过的知识串联起来，并结合一些算法和思想使读者可以学会将 C++语言的基础知识和应用结合起来。通过本章，可以掌握开发小游戏的设计思路。

17.3 吃豆子游戏

在项目中，主要的文件包含如下几个。

- ☑ GMap.h：地图类的声明文件。
- ☑ GMap.cpp：地图类的实现文件。
- ☑ GObject.h：物体类的声明文件。
- ☑ GObject.cpp：物体类的实现文件。
- ☑ pacman.cpp：创建主窗口，实现游戏运行的客户端。

17.3.1 PacMan 程序框架初步分析

制作软件之前，都需要对需求作较为全面的分析。

在吃豆子（PacMan）游戏中，玩家可以操作的角色是一张“大嘴”。游戏的目的是操作“大嘴”躲避敌人吃掉所有的豆子。游戏中物体所在的场地是二维的平面，并且存在墙与障碍物，游戏的效果如图 17.17 所示。

图中弧形的物体形象地体现了大嘴的角色，其他彩色轮廓类似“章鱼”的物体则是敌人。在过道中充斥着的小圆圈是豆子。使用键盘的方向键对游戏操作，大嘴路过可以“吃掉”豆子，但触碰到敌人则会失败。

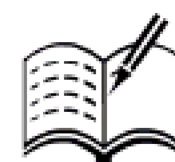
以面向对象的设计方法来思考，吃豆子中的所有会移动的物体具有很多共同的特性，将它们列举出来：

- ☑ 会移动。
- ☑ 移动分为上、下、左、右 4 个方向。
- ☑ 碰到墙和障碍物会停止。
- ☑ 物体拥有自己的坐标。

在某些方面，它们不同，但所处的层面是相同的：

- ☑ 拥有绘图效果，但各部相同。
- ☑ 各个物体都有自己的移动准则。例如，“大嘴”是玩家控制的，而敌人则是依照设定好的人工智能行动。





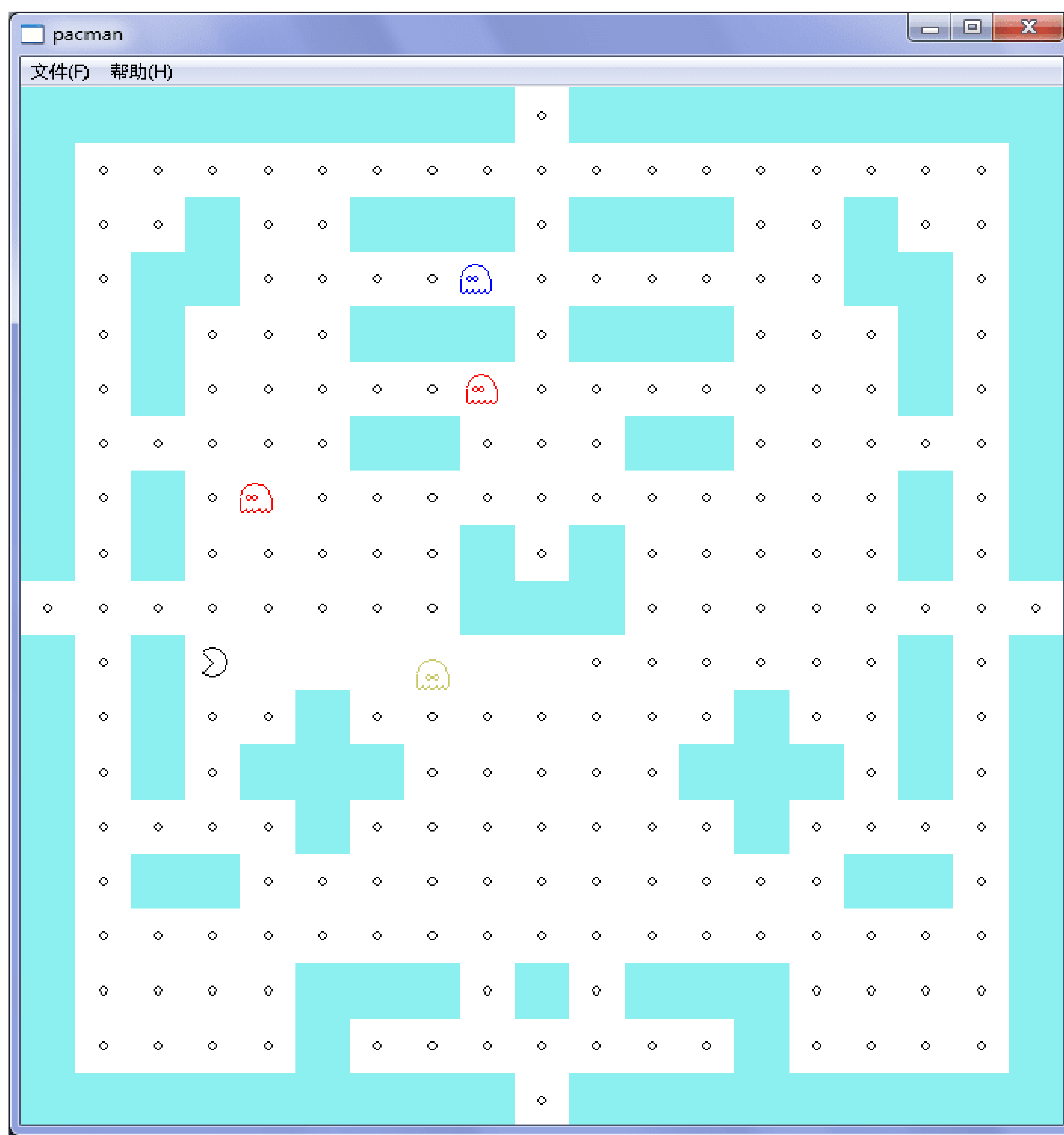


图 17.17　吃豆子效果图

这些物体完全不同的方面有：

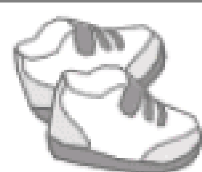
☑　大嘴能吃豆子，敌人不能。

☑　敌人能够“抓住”大嘴，大嘴不能够“抓住”任何物体。

依照以上列出的条件，可以设计一个物体类，本项目中取名为 GObject。它是游戏中可移动物体的父类。将共性放入父类中，则该类应该具有的属性有：在平面地图上所处位置、自身的朝向、碰撞检测。现在用代码描述这个父类（即声明成员）并建立一个方向枚举：

```
enum TWARDS{UP,DOWN,LEFT,RIGHT,OVER};                    //方向枚举
class Gobject
{
protected:
    TWARDS tw                                            //朝向
    //相同的特性:
    POINT point;                                         //中心坐标
    bool Collision() ;                                   //逻辑碰撞检测，将物体摆放到合理的位置
    //相同的特性，但是实现方法不同:
    void virtual action() = 0;                           //具体的行为
    void virtual Draw( HDC& hdc)=0;                      //绘制对象
};
```

也许读者已经开始阅读本章的代码，父类所描述内容要多一些。原因是我们还未讨论过游戏实现的细节。那么，现在就从碰撞检测的实现开始讨论吧。





在游戏中如何让程序知道物体在撞墙？通过物体所在点的位置和墙体边缘位置的检测似乎是个好主意。计算方法可以是中心坐标和朝向所对应的墙的位置与物体的宽度作比较，若是大于宽度，则没有碰上。这种碰撞检测方法虽然趋于真实，但是实现非常复杂。首先需要记录所有墙体边缘点的坐标（像素点），然后行走一步就判断一下朝向是否撞墙。其中还需要找到相应方向的墙壁，否则需要便利所有墙壁边缘点的坐标。实现之后的情况可能还会存在一些不希望被观测到的结果，例如转弯之后没有完全半截身体卡在墙中等。

那么，如何设计吃豆子中的碰撞检测呢？地图的记录方式是关键。地图的记录数据量越少，计算的方法越简单。将整个地图分为若干个正方形的小方格，物体每到一个方格才会去做碰撞检测。地图记录的是这些小方格是否有墙体与豆子的信息，物体通过地图来判断是否撞到了墙壁。

本项目所设计的地图为长、高各 19 个方格。使用一个二维矩阵来记录它，同时地图还应该记录方格的大小。设计一个地图类：

```
class Gmap
{
protected:
        static int LD ;                                         //障碍物尺寸
        bool mapData[MAPLENTH ][MAPLENTH ];                     //障碍物逻辑地图点阵
frined class GObject;                                           //将自身数据提供给友元物体类
};
```

其中，MAPLENTH 是数字 19 的宏。LD 是墙的尺寸。因为地图类所有自身、自类对象的墙壁大小应该相同，所以应该将其设定为静态变量。

现在拥有了地图类，那么对于所有物体而言，它们使用的地图也应该是同一张。向 GObject 类添加地图类指针变量：

```
protected:
        static GMap* pStage; //指向地图类的指针，设置为静态，使所有自类对象都能够使用相同的地图
```

这样物体类就包含了地图的信息。在物体类中的碰撞检测依照地图类所记录方格信息进行判断。物体行进到一个方格中，判断它当前朝向的下一个格子是否有墙壁是当前碰撞的判断标准。在此之前需要判断的是物体是否在方格的中央，检测的标准如图 17.18 所示。

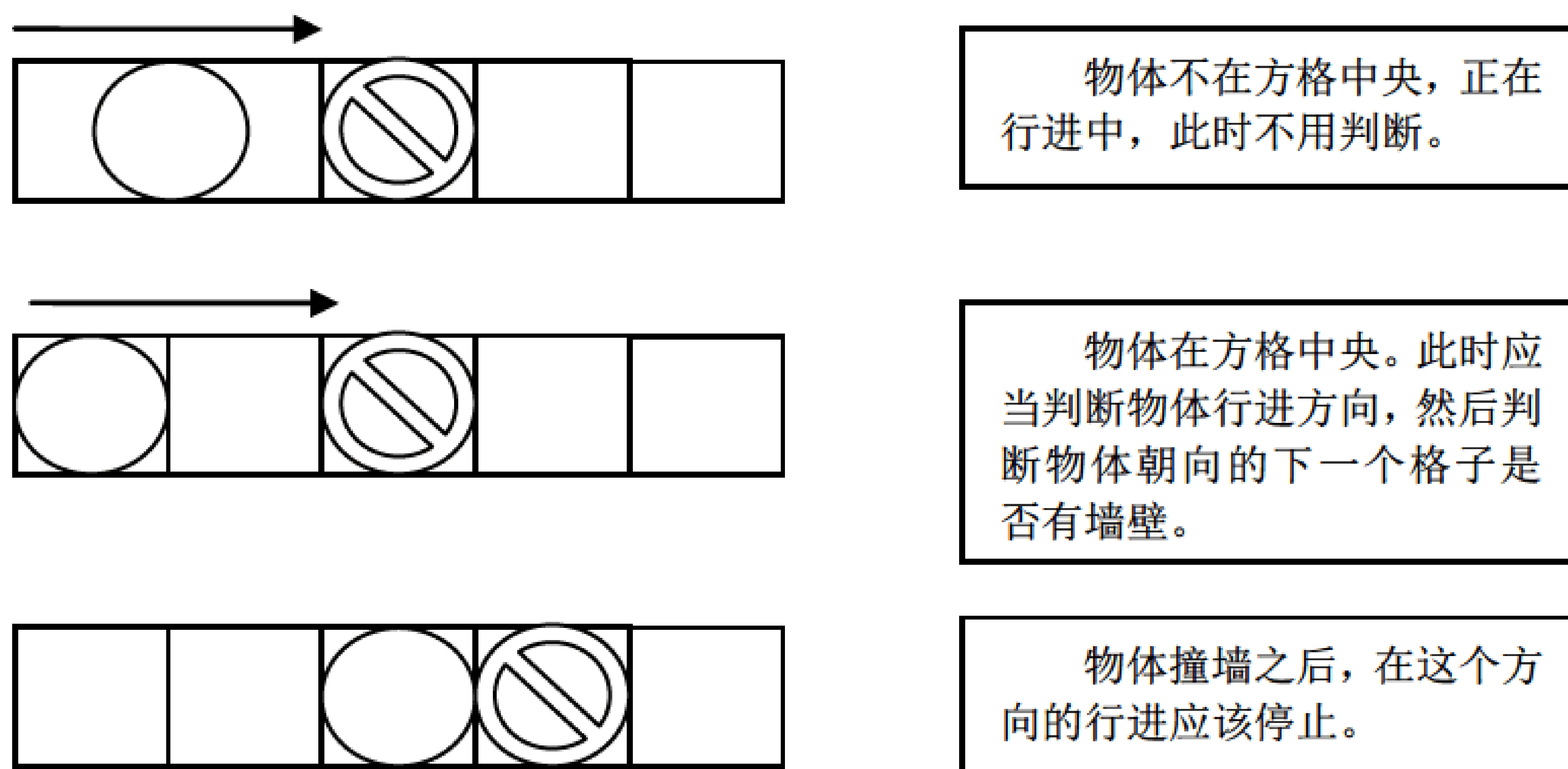
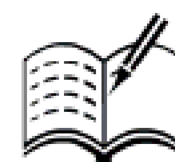


图 17.18　碰撞检测标准





现在需要记录物体在地图矩阵中的当前位置，以及计算物体是否在方格中。在 GObject 类中声明以下成员：

```
protected:
        bool Achive();                          //判断物体是否到达逻辑坐标位置
        int dRow;                               //逻辑横坐标（即所在矩阵的行）
        int dArray;                             //逻辑纵坐标（即所在矩阵的列）
        int speed;                              //速度
        virtual void AchiveCtrl();              //到达逻辑点后更新数据
```

将更新数据函数 AchiveCtrl 设定为虚函数。这个函数的功能是在判断物体到达格子后，更新物体在矩阵中的行列坐标。玩家所操作的“大嘴”在到达方格后需要判断是否消除了豆子，这样等于在到达格子后添加了一种行为，这样很适合将此函数设置为虚函数供子类使用。

物体到达格子后除了向以前的方向前行，还可以转弯。在游戏中并不提倡随时都可以转弯，因为大多数情况一定是无效的指令（由于撞墙），所以在碰撞检测中也应该包含方向的更新和指令的有效性。“大嘴”和敌人都应该存在着方向指令，这样才能够在地图上转弯。在物体类里可以使用一个前面定义的方向枚举 TWARDS 类型来储存这个指令。

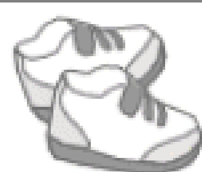
```
TWARDS twCommand;                               //指令缓存
```

17.3.2　碰撞检测的实现

在地图类 GMap 类中，使用了 bool 型的二维矩阵储存地图上墙壁位置的信息。当值为 false 时，则表示该位置有墙壁；当值为 true 时，说明该位置没有墙壁。

依照碰撞检测的标准，首先应该更新物体所在的行、列数据。

```
bool GObject::Achive()
{
         int n =(point.x- pStage->LD/2)%pStage->LD;
         int k =(point.y- pStage->LD/2)%pStage->LD;
          bool l = (n==0&&k==0);
          return l;
}
void GObject::AchiveCtrl()
{
        if(Achive())
        {
                dArray = PtTransform(point.x);          //更新列
                dRow = PtTransform(point.y);            //更新行
        }
}
int GObject::PtTransform(int k)
{
        return (k -(pStage->LD)/2)/pStage->LD;
}
```





首先，讲解 PtTransform 这个函数。在类中将它声明为访问权限为 protected 的函数，它的作用是将物体在屏幕上的坐标转换为行/列坐标。

以图 17.19 左上角第一个格子为基准。

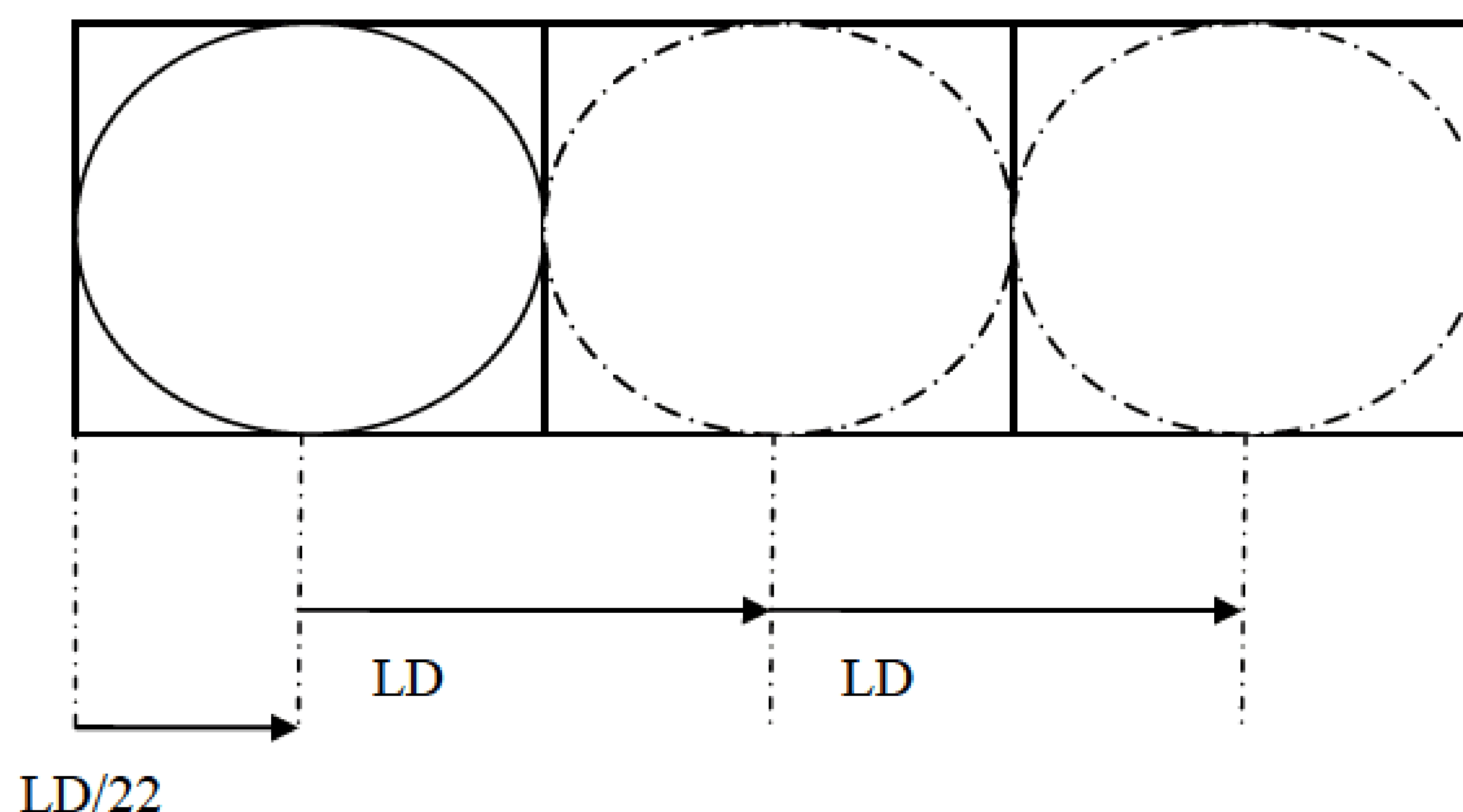


图 17.19　坐标偏移与转换

第一个格子的左上角坐标为 0。以列为例，当换算方格中心坐标与第一个方格的距离时，需要先减去第一个方格的左边界与中心坐标的距离，然后整除方格大小。

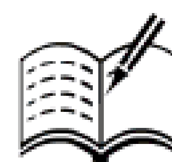
同样地，在 Archive 函数中判断物体是否到达方格的判断条件是：查看方格的中心是否和物体中心重合，应用矩阵坐标与窗口坐标的转换。

函数 ArchiveCtrl 的内容是当物体到达方格中心时，更新物体的行/列坐标。

完成了坐标更新之后，还要查看当前指令的有效性。这个指令的存在是在物体到达方格之前产生的，在物体到达方格时进行判断。在碰撞检测函数 Collision 中，应当先填写下列代码：

```
bool b = false;
AchiveCtrl();//更新行、列的数据若是大嘴，则会执行 PacMan 重写的 AchiveCtrl 函数消除豆子
//判断指令的有效性
if(dArray<0||dRow<0||dArray>MAPLENTH||dRow>MAPLENTH)
{
	b = true;
}
else if(Achive())
{
	switch(twCommand)                                        //判断行进的方向
	{
		case LEFT:
			if(dArray>0&&!pStage->mapData[dRow][dArray-1])  //判断下一个格子是否能够通行
			{
				b = true;                                //指令无效
			}
			break;
			//以下方向的判断原理相同
		case RIGHT:
			if(dArray<MAPLENTH-1&&!pStage->mapData[dRow][dArray+1])
			{
				b = true;
			}
```



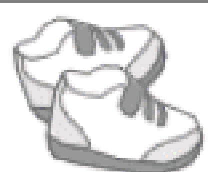


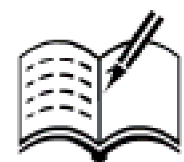
```
            break;
        case UP:
            if(dRow>0&&!pStage->mapData[dRow-1][dArray])
            {
                b = true;
            }
            break;
        case DOWN:
            if(dRow<MAPLENTH-1&&!pStage->mapData[dRow+1][dArray])
            {
                b = true;
            }
            break;
        }
    if(!b)
    {
        tw =twCommand;                                    //没撞墙，指令成功
    }
  }
}
```

以上是碰撞检测代码中检测方向指令有效性的片断。首先更新行列，若物体在屏幕外，则不可以改变指令。之后判断到达方格的物体指令方向的下一个格子是否有墙存在。若有墙存在，则指令无效，若没有撞墙则指令成功，将方向替换成指令的方向。参数 b 的作用除了判断指令是否有效外，还会在人工智能的设定用到。稍后将加以说明。

之后物体应该朝着当前的方向继续前行。当下一个格子有墙壁出现，物体不随速度 speed 改变位置。

```
switch(tw)//判断行进的方向
    {
        case LEFT:
            if(dArray>0&&!pStage->mapData[dRow][dArray-1])  //判断下一个格子是否能够通行
            {
                b= true;
                break;                                //“撞墙了”
            }
            if(point.x<MIN)
            {
                point.x = MAX;
            }
            point.x -= speed;
            break;
            //以下方向的判断原理相同
        case RIGHT:
            if(dArray<MAPLENTH-1&&!pStage->mapData[dRow][dArray+1])
            {
                b= true;
                break;                                //“撞墙了”
            }
```





Note

```
				point.x += speed;
				if(point.x>MAX)
				{
					point.x = MIN;
				}
				break;
			case UP:
				if(dRow>0&&!pStage->mapData[dRow-1][dArray])
				{
					b= true;
					break;						// "撞墙了"
				}
				if(point.y<MIN)
				{
					point.y = MAX;
				}
				point.y -=speed;
				break;
			case DOWN:
				if(dRow<MAPLENTH-1&&!pStage->mapData[dRow+1][dArray])
				{
					b= true;
					break;						// "撞墙了"
				}
				point.y +=speed;
				if(point.y>MAX)
				{
					point.y = MIN;
				}
				break;
		}
	return b;
```

无须担心物体会因此卡在两个格子中间，不能行动。因为在此之前已经更新过行列数据——只有在物体到达格子中央才更新。MAX 和 MIN 分别代表超出地图边界一个方格的位置。当物体超过地图边界到达地图外则会从另一边出现。

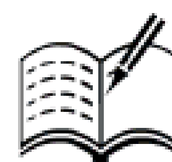
至此物体在地图中的碰撞检测已经设定完毕。而当前需要一张地图来配合物体类使用。下面将讨论地图类的设计。

17.3.3　地图类的设计

若我们想进行一个多关卡的 PacMan 游戏，那么它的地图一定不止一张。有多种办法设计这项功能。可以创建一个存放地图矩阵的容器（数组、链表、STL 模板库容器）。它的好处是以简便的方式存放地图，更换地图也很简便，只需要将当前的地图数据更换到容器一个位置的地图数据。

创建地图不只是通过计算来实现的，最好可以通过可视化工具来使我们看到做的地图是什么





样子的，而不是用编译器一遍又一遍地调试来查看地图。

二维数组可以用列表的方式来初始化，例如：

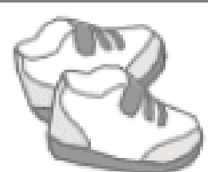
```
#define A true
#define B false
bool Stage_1::initData[MAPLENTH][MAPLENTH]=
                                    {
                                      B,B,B,B,B,B,B,B,B,A,B,B,B,B,B,B,B,B,B,//0
                                      B,A,A,A,A,A,A,A,A,A,A,A,A,A,A,A,A,A,B,//1
                                      B,A,A,B,A,A,B,B,B,A,B,B,B,A,A,B,A,A,B,//2
                                      B,A,B,B,A,A,A,A,A,A,A,A,A,A,A,B,B,A,B,//3
                                      B,A,B,A,A,A,B,B,B,A,B,B,B,A,A,A,B,A,B,//4
                                      B,A,B,A,A,A,A,A,A,A,A,A,A,A,A,A,B,A,B,//5
                                      B,A,A,A,A,A,B,B,A,A,A,B,B,A,A,A,A,A,B,//6
                                      B,A,B,A,A,A,A,A,A,A,A,A,A,A,A,A,B,A,B,//7
                                      B,A,B,A,A,A,A,A,B,A,B,A,A,A,A,A,B,A,B,//8
                                      A,A,A,A,A,A,A,A,B,B,B,A,A,A,A,A,A,A,A,//9
                                      B,A,B,A,A,A,A,A,A,A,A,A,A,A,A,A,B,A,B,//10
                                      B,A,B,A,A,B,A,A,A,A,A,A,A,B,A,A,B,A,B,//11
                                      B,A,B,A,B,B,B,A,A,A,A,A,B,B,B,A,B,A,B,//12
                                      B,A,A,A,A,B,A,A,A,A,A,A,A,B,A,A,A,A,B,//13
                                      B,A,B,B,A,A,A,A,A,A,A,A,A,A,A,B,B,A,B,//14
                                      B,A,A,A,A,A,A,A,A,A,A,A,A,A,A,A,A,A,B,//15
                                      B,A,A,A,A,B,B,B,A,B,A,B,B,B,A,A,A,A,B,//16
                                      B,A,A,A,A,B,A,A,A,A,A,A,A,B,A,A,A,A,B,//17
                                      B,B,B,B,B,B,B,B,B,A,B,B,B,B,B,B,B,B,B,//18
                                      };
#undef A
#undef B
```

也许印刷体的格式会让列不太齐整，在编译器中矩阵列表各行和列是对齐的。可以通过这种初始化来达到观测各行各列的目的。这张地图相应的 bool 变量已经被设定成 A、B，分别代表 true 和 false。

设置多关卡的地图的另外一种方法是可以建立一个地图类，它本身不存放地图数据。自类地图分别使用静态的地图矩阵初始化内部的地图矩阵成员 mapData，还可以采用不同的颜色类型成员变量来绘制地图。

以下是基类 GMap 与一个关卡 Stage_1 在头文件中的全部声明：

```
#pragma once
#include "stdafx.h"
#include <list>
#define   MAPLENTH 19                                          //逻辑地图大小
#define P_ROW 10
#define P_ARRAY 9
#define E_ROW 8
#define E_ARRAY 9
using std::list;
//抽象类 GMap
```






```
class GMap{
protected:
	static int LD ;						//障碍物尺寸
	static int PD;						//豆子的半径
	void InitOP();						//敌我双方出现位置没有豆子出现
	bool mapData[MAPLENTH ][MAPLENTH ];			//障碍物逻辑地图点阵
	bool peaMapData[MAPLENTH ][MAPLENTH ];			//豆子逻辑地图点阵
	COLORREF color;
public:
	void	DrawMap(HDC& hdc);			//绘制地图
	void	DrawPeas(HDC& hdc);			//绘制豆子
	virtual ~GMap();
	GMap(){
		}
friend class GObject;					//允许物体类使用直线的起点和终点的信息做碰撞检测
friend class PacMan;					//允许“大嘴”访问豆子地图
};
// “第一关”
class Stage_1:public GMap
{
private:
	 bool static initData[MAPLENTH][MAPLENTH];
public:
	Stage_1();
};
```

豆子地图也是地图矩阵，豆子的位置在地图中和墙壁的位置是恰好互补的。

在 Gmap 类的实现文件中，初始化自身的静态变量，实现 InitOP 函数。

```
int GMap::LD =36;					//地图方格大小
int GMap::PD =3;					//豆子的绘图半径
//敌我双方出现位置没有豆子出现
void GMap::InitOP()
{
	peaMapData[E_ARRAY][E_ROW] = false;
	peaMapData[P_ARRAY][P_ROW] = false;
}
```

InitOP 函数将敌人和“大嘴”最初位置的豆子去掉。

在 Stage_1 类中使用静态矩阵初始化自身的成员矩阵。

```
Stage_1::Stage_1()
{
	color =RGB(140,240,240);
	for(int i= 0;i<MAPLENTH;i++)
	{
		for(int j =0;j<MAPLENTH;j++)
		{
			this->mapData[i][j] = this->initData[i][j];
			this->peaMapData[i][j] =initData[i][j];
```



```
            }
        }
//敌我双方出现位置没有豆子出现
        peaMapData[10][10] = true;
        this->InitOP();
}
```



17.3.4　数据更新

在 Visual Studio 2010 的类视图中分别选中 BlueOne、Enermy、GObject、PacMan、RedOne 和 YellowOne，单击图 17.20 红圈位置的查看类视图就可以查看到程序中 GObject 和子类的继承关系。

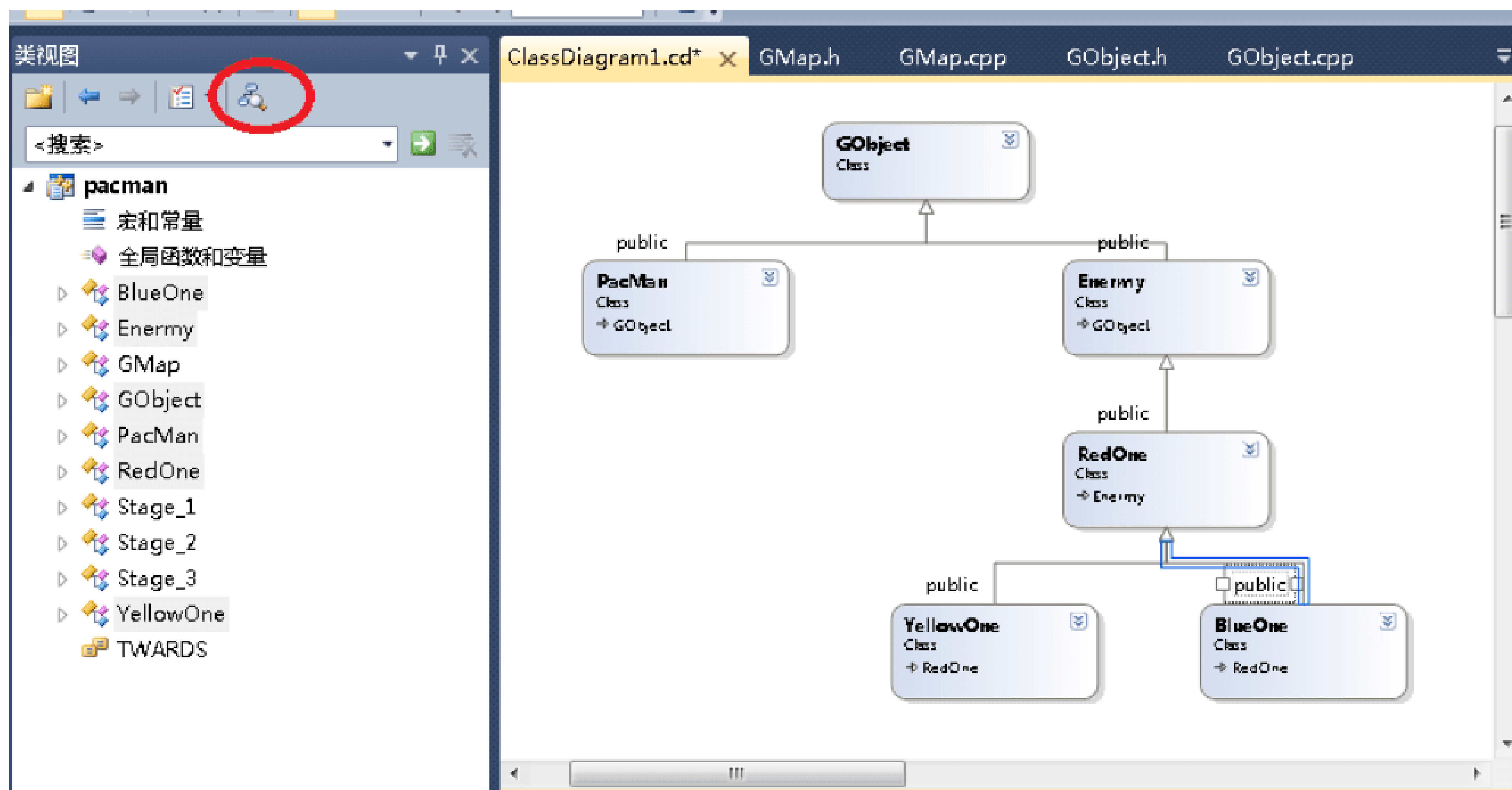


图 17.20　Visual Studio 类图

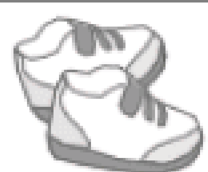
PacMan 类是“大嘴”的实现类，它直接继承于 GObject 类。Enermy 是敌人类的父类，同样也直接继承于父类 GObject。

在窗口应用程序中，我们希望使用一个函数就可以代表这个类所有的数据变化。在父类中定义 Action 函数，它的实现就是数据的变化。

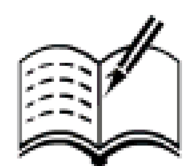
```
public:
        void virtual action() = 0;//数据变更的表现
```

它是一个纯虚函数。它阻止了物体类的实例化，因为物体类是敌我双方的“模板”，不可能被实例化。子类的数据变化方式不会一样，而父类并不需要具体去规定自类的行为方式。所以将它设定成纯虚函数。

在 PanMan 中的行为方式只有碰撞检测 Collison。



```
void PacMan::action()
{
	Collision();
}
```



Note

在 Enermy 类中，应该具备一个函数来实现人工智能的实现方法。将它声明为：

```
void virtual MakeDecision(bool b) = 0;//AI 实现
```

在吃豆子游戏中，人工智能唯一能改变的只有方向指令。在 Enermy 的 3 个子类中，MakeDecision 只能更改方向指令，而不能更改和地图位置有关的数据。敌人需要“知晓”大嘴的位置信息，才能够作出相应的动作。在 Enermy 类内部声明一个 PacMan 类的指针：

```
public:
	static PacMan* player;
```

静态变量必须初始化，需要在类的外部将它初始化为空：

```
PacMan* Enermy::player   = NULL;
```

“大嘴”所在的行和列的数据应该是“公开”的，在 GObject 提供访问权限为 public 的 GetRow 和 GetArray 函数，用来获得行、列的信息。

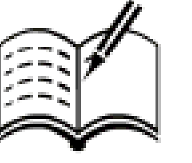
```
int GObject::GetRow()
{
	return dRow;
}
int GObject::GetArray()
{
	return dArray;
}
```

所有敌人都共用同一个“大嘴”的数据。

子类们的 AI（人工智能）实现如下：

```
void RedOne::MakeDecision(bool b)
{
	int i = rand();
if(b)//撞到墙壁，改变方向
{
	//逆时针转向
	if(i%4==0)
	{
		tw == UP?twCommand = LEFT:twCommand=UP;
	}
	else if(i%3==0)
	{
		tw == DOWN?twCommand =RIGHT:twCommand=DOWN;
	}
```





```
        else if(i%2==0)
        {
            tw == RIGHT?twCommand = UP:twCommand=RIGHT;
        }
        else
        {
            tw == LEFT?twCommand = DOWN:twCommand=LEFT;
        }
        return;
    }
        if(i%4==0)
        {
            twCommand!=UP?tw==DOWN:twCommand ==UP;
        }
        else if(i%3==0)
        {
            tw != DOWN?twCommand = UP:twCommand=DOWN;
        }
        else if(i%2==0)
        {
            tw != RIGHT?twCommand = LEFT:twCommand=RIGHT;
        }
        else
        {
            tw != LEFT?twCommand = RIGHT:twCommand=LEFT;
        }
    }
    void BlueOne::MakeDecision(bool b)
    {
        const int DR = this->dRow-player->GetRow();
        const int DA = this->dArray-player->GetArray();
        if(!b&&DR==0)
        {
            if(DA<=BLUE_ALERT&&DA>0)                    //玩家在左侧边警戒范围
            {
                twCommand = LEFT;                       //向左移动
                return;
            }
            if(DA<0&&DA>=-BLUE_ALERT)                   //右侧警戒范围
            {
                twCommand = RIGHT;                      //向右移动
                return;
            }
        }
        if(!b&&DA==0)
        {
            if(DR<=BLUE_ALERT&&DR>0)                    //下方警戒范围
            {
                twCommand = UP;
                return;
```


Note

```
        }
        if(DR<0&&DR>=-BLUE_ALERT)                       //上方警戒范围
        {
            twCommand = DOWN;
            return;
        }
    }
    RedOne::MakeDecision(b);                            //不在追踪模式时 RED 行为相同
}
void YellowOne::MakeDecision(bool b)
{
    const int DR = this->dRow-player->GetRow();
    const int DA = this->dArray-player->GetArray();
    if(!b)
    {
        if(DR*DR>DA*DA)
        {
            if(DA>0)                                    //玩家在左侧边警戒范围
            {
                twCommand = LEFT;                       //向左移动
                return;
            }
            else if(DA<0)                               //右侧警戒范围
            {
                twCommand = RIGHT;                      //向右移动
                return;
            }
        }
        else
        {
            if(DR>0)                                    //下方警戒范围
            {
                twCommand = UP;
                return;
            }
            if(DR<0)                                    //上方警戒范围
            {
                twCommand = DOWN;
                return;
            }
        }
    }
    RedOne::MakeDecision(b);
}
```

以上 3 种敌人从上往下的行动模式分别为松散型、守卫型和扰乱型。b 代表的是执行碰撞检测后的返回结果，true 代表撞墙，false 代表没有撞墙。

☑ 松散型：使用了 rand 函数产生了一个随机数，根据随机数来判定方向指令。但是它在撞到墙壁之前不会突然回头，在撞墙之后返回的概率很大，程序中会把撞墙方向修改为



反方向。

☑ 守卫型：只有当“大嘴”与它处于同一行或列的警戒范围时才能“察觉”并追踪。BLUE_ALERT 是一个整型常数宏，稍后会列出它。不在警戒状态时，它的行动模式和松散型相同。

☑ 扰乱型：不断地接近“大嘴”，但不会在空旷的区域上主动抓捕。在撞到墙壁后变为松散型行动模式。



物体类中碰撞检测只约束了物体移动与墙壁的关系。“大嘴”需要吃豆子，敌人则需要有抓捕“大嘴”的具体实现。

“大嘴”需要将豆子地图中的相应格子的数据变为 false，发生的时机在到达格子中心处。在 PacMan 类中重写 AchiveCtrl 函数。

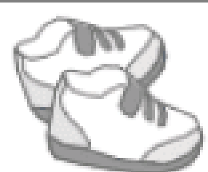
```
void PacMan::AchiveCtrl()
{
	GObject::AchiveCtrl();                              //实现物体类更新行列的功能
	if(Achive())
	{
		if(dRow>=0&&dRow<MAPLENTH&&dArray>=0&&dArray< MAPLENTH)    //防止数组越界
		{
			if(pStage->peaMapData[dRow][dArray])
			{
				pStage->peaMapData[dRow][dArray] = false;
			}
		}
	}
}
```

定义一个访问权限为 public 的成员函数。当地图中没有豆子时，玩家获得此关的胜利：

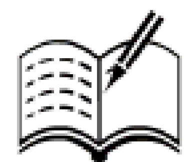
```
bool PacMan::Win()
{
	for(int i=0;i<=MAPLENTH;i++)
	{
		for(int j=0;j<=MAPLENTH;j++)
		{
			if(pStage->peaMapData[i][j]==true)
			{
				return false;                     //存在任意一个豆子，没取得胜利
			}
		}
	}
	return true;                                  //没有豆子，胜利
}
```

敌人 Enermy 类中增加一个函数 Catch 用来抓捕“大嘴”。PacMan 类提供一个 Over 方法，将自身的方向变为 OVER，表示失败。

```
void PacMan::Over()
{
```



```
        tw = OVER;
}
```



Note

在 PacMan 中实现一个提供自身位置的函数，访问权限为 public：

```
POINT PacMan::GetPos()
{
        return point;
}
```

这样在 Enermy 中就可依照“大嘴”位置实施抓捕。实现访问权限为 protected 的 Catch 函数。

```
void Enermy::Catch()
{
        int DX =point.x -player->GetPos().x;
        int DY =point.y -player->GetPos().y;
        if((-RD<DX&&DX<RD)&&(-RD<DY&&DY<RD))
        {
                player->Over();
        }
}
```

RD 代表的是物体类的绘图范围，是一个整型常数宏。当敌我双方中心点的距离小于绘图范围，那么“大嘴”就被抓到了。

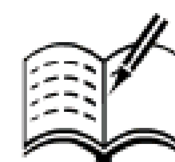
敌人类的行动模式可以分为 3 步，实现如下：

```
void Enermy::action()
{
        bool b = Collision();
        MakeDecision(b);
        Catch();
}
```

这样就完成了敌我数据的更新步骤。下面将完整的 GObject 头文件列出：

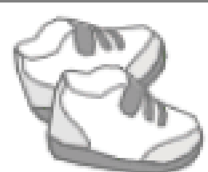
```
#include "stdafx.h"
#include "GMap.h"
#define PLAYERSPEED 6                                   //玩家速度
#define ENERMYSPEED 4                                   //敌人速度
#define LEGCOUNTS 5                                     //敌人腿的数量
#define DISTANCE 10                                     //图形范围
#define BLUE_ALERT 8                                    //蓝色警戒范围
#define D_OFFSET    2                                   //绘图误差
#define RD (DISTANCE + D_OFFSET)                        //绘图范围
#include <time.h>
enum TWARDS{UP,DOWN,LEFT,RIGHT,OVER};                   //方向枚举
class GObject{                                          //物体类
protected:
        int mX;
        int mY;
```

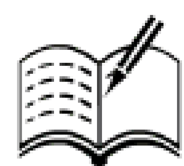




```
        TWARDS twCommand;                         //指令缓存
        POINT point;                              //中心坐标
        int dRow;                                 //逻辑横坐标
        int dArray;                               //逻辑纵坐标
        int speed;                                //速度
        TWARDS tw;                                //朝向
        int frame;                                //帧数
        //子程序
        bool Achive();                            //判断物体是否到达逻辑坐标位置
        bool Collision() ;                        //逻辑碰撞检测，将物体摆放到合理的位置
        int PtTransform(int k);                   //将实际坐标转换为逻辑坐标
        virtual void AchiveCtrl();                //到达逻辑点后更新数据
public:
        void SetPosition(int Row,int Array);
        void DrawBlank( HDC& hdc);
        void virtual Draw( HDC& hdc)=0;           //绘制对象
        static GMap* pStage;        //指向地图类的指针，设置为静态，使所有自类对象都能够使用相同的地图
        GObject(int Row,int Array)
        {
                frame = 1;
                pStage = NULL;
                this->dRow = Row;
                this->dArray   = Array;
                this->point.y = dRow*pStage->LD+pStage->LD/2;
                this->point.x = dArray*pStage->LD+pStage->LD/2;
                this->mX =point.x;
                this->mY =point.y;
        }
        void virtual action() = 0;                //数据变更的表现
        int GetRow();
        int GetArray();
};
//大嘴，玩家控制的对象
class PacMan:public Gobject
{
protected:
        virtual void AchiveCtrl();                //重写虚函数

public:
        POINT GetPos();
        TWARDS GetTw();
        bool Win();
        void Draw(HDC& hdc);
        void SetTwCommand(TWARDS command);
        void Over();
        PacMan(int x,int y):GObject(x,y)
        {
                this->speed = PLAYERSPEED;
                twCommand=tw = LEFT;
        }
```





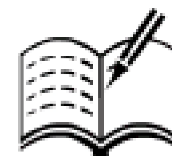
Note

```
    void action();
};
//追捕大嘴的敌人
class Enermy:public Gobject
{
protected:
    void Catch();                                         //是否抓住大嘴
    void virtual MakeDecision(bool b) = 0;                //AI 实现
    COLORREF color;
public:
    static PacMan* player;
    void virtual   Draw(HDC& hdc);
    Enermy(int x,int y):GObject(x,y)
    {
        this->speed = ENERMYSPEED;
        tw = LEFT;
        twCommand = UP;
    }
    void virtual action();
};
class RedOne:public Enermy                                //随机移动
{
protected:
    void virtual MakeDecision(bool b);
public:
    void Draw(HDC& hdc);
    RedOne(int x,int y):Enermy(x,y)
    {
        color = RGB(255,0,0);
    }
};
class BlueOne:public RedOne                               //守卫者
{
protected:
    void virtual MakeDecision(bool b);
public:
    void Draw(HDC& hdc);
    BlueOne(int x,int y):RedOne(x,y)
    {
        color = RGB(0,0,255);
    }
};
class YellowOne:public RedOne                             //扰乱者
{
protected:
    void virtual MakeDecision(bool b);
public:
    void Draw(HDC& hdc);
    YellowOne(int x,int y):RedOne(x,y)
    {
```



```
            color = RGB(200,200,100);
        }
};
```

剩下的工作就是绘制地图和物体。



Note

17.3.5　绘图

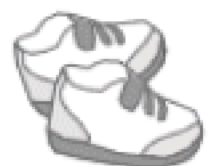
首先完成绘制地图的工作。地图元素主要有两种：空地和墙。空地的颜色使用窗口背景的白色，墙体可以使用 FillRect 函数将墙壁位置的方格着色。

实现 GMap 的 DrawMap 函数：

```
void GMap::DrawMap(HDC& memDC)
{
    for(int i = 0;i<MAPLENTH;i++)
    {
        for(int j = 0;j<MAPLENTH;j++)
        {
            if(!mapData[i][j])                          //绘制墙壁
            {
                RECT rect;
                rect.left = j*LD;
                rect.top = i*LD;
                rect.right = (j+1)*LD;
                rect.bottom = (i+1)*LD;
                FillRect(memDC,&rect,CreateSolidBrush(color));
            }
        }
    }
}
```

这样就完成了绘制墙壁的工作。地图类中还包含豆子位置的信息，DrawPeas 函数使用豆子地图的信息将豆子绘制到屏幕上：

```
void GMap::DrawPeas(HDC& hdc)
{
    for(int i = 0;i<MAPLENTH;i++)
    {
        for(int j = 0;j<MAPLENTH;j++)
        {
            if(peaMapData[i][j])
            {
Ellipse(hdc,(LD/2-PD)+j*LD,(LD/2-PD)+i*LD,(LD/2+PD)+j*LD,(LD/2+PD)+i*LD);
            }
        }
    }
}
```



在 PacMan 类中的 AchiveCtrl 函数中会改变豆子地图信息，相应位置的元素会设置为 false。在绘制豆子时，绘制函数会通过更改后的 peaMapData 数据绘制豆子。

物体类 GOject 的绘图函数 Draw 是一个纯虚函数。GObject 中声明的 frame 变量代表帧数，它的概念有些像翻页动画中的页数。将书页连续翻动，就形成了动画。



Note

敌人的头部是半圆形，身体由两条直线构成；敌人具有若干个半圆形的腿部，依照方向会挪动眼睛。需要随着帧数改变的是腿部。眼睛的方向会随着它行进的方向而发生改变，看起来像是在关注前方。

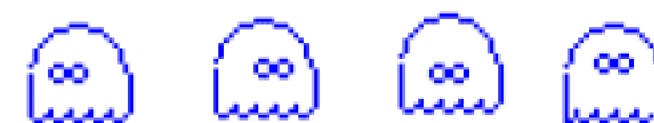
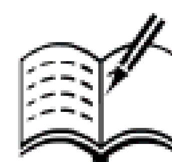


图 17.21　敌人注视的方向

绘制敌人的代码：

```
void Enermy::Draw(HDC& hdc)
{
        HPEN pen =::CreatePen(0,0,color);
        HPEN oldPen = (HPEN)SelectObject(hdc,pen);
        Arc(hdc,point.x-DISTANCE,point.y-DISTANCE,
                point.x+DISTANCE,point.y+DISTANCE,
                point.x+DISTANCE,point.y,
                point.x-DISTANCE,point.y);                                  //半圆形的头
        int const LEGLENTH = (DISTANCE)/(LEGCOUNTS);
        //根据帧数来绘制身体和“腿部”
        if(frame%2 == 0)
        {
                MoveToEx(hdc,point.x-DISTANCE,point.y,NULL);                //矩形的身子
                LineTo(hdc,point.x-DISTANCE,point.y +DISTANCE - LEGLENTH);
                MoveToEx(hdc,point.x+DISTANCE,point.y,NULL);
                LineTo(hdc,point.x+DISTANCE,point.y +DISTANCE - LEGLENTH);
                for(int i = 0;i<LEGCOUNTS;i++)                              //从左往右绘制“腿部”
                {
                        Arc(hdc,point.x-DISTANCE+i*2*LEGLENTH,point.y+DISTANCE-2*LEGLENTH,
                                point.x-DISTANCE+(i+1)*2*LEGLENTH,point.y+DISTANCE,
                                point.x-DISTANCE+i*2*LEGLENTH,point.y+DISTANCE-LEGLENTH,
                                point.x-DISTANCE+(i+1)*2*LEGLENTH,point.y+DISTANCE-LEGLENTH);
                }
        }
        else{
                MoveToEx(hdc,point.x-DISTANCE,point.y,NULL);                //绘制身体
                LineTo(hdc,point.x-DISTANCE,point.y +DISTANCE);
                MoveToEx(hdc,point.x+DISTANCE,point.y,NULL);
                LineTo(hdc,point.x+DISTANCE,point.y +DISTANCE);
                //从左往右绘制“腿部”
                MoveToEx(hdc,point.x-DISTANCE,point.y+DISTANCE,NULL);
                LineTo(hdc,point.x-DISTANCE+LEGLENTH,point.y+DISTANCE-LEGLENTH);
                for(int i = 0;i<LEGCOUNTS-1;i++)
                {
                        Arc(hdc,point.x-DISTANCE+(1+i*2)*LEGLENTH,point.y+DISTANCE-2*LEGLENTH,
                                point.x-DISTANCE+(3+i*2)*LEGLENTH,point.y+DISTANCE,
                                point.x-DISTANCE+(1+i*2)*LEGLENTH,point.y+DISTANCE-LEGLENTH,
                                point.x-DISTANCE+(3+i*2)*LEGLENTH,point.y+DISTANCE-LEGLENTH);
                }
```





Note

```
            MoveToEx(hdc,point.x+DISTANCE,point.y+DISTANCE,NULL);
            LineTo(hdc,point.x+DISTANCE-LEGLENTH,point.y+DISTANCE-LEGLENTH);
        }
        //根据方向绘制眼睛
        int R = DISTANCE/5;                                    //眼睛的半径
        switch(tw)
        {
            case UP:
            Ellipse(hdc,point.x-2*R,point.y-2*R,
                point.x,point.y);
            Ellipse(hdc,point.x,point.y-2*R,
                point.x+2*R,point.y);
            break;
        case DOWN:
            Ellipse(hdc,point.x-2*R,point.y,point.x,point.y+2*R);
            Ellipse(hdc,point.x,point.y,point.x+2*R,point.y+2*R);
            break;
        case LEFT:
            Ellipse(hdc,point.x-3*R,point.y-R,
                point.x-R,point.y +R);
            Ellipse(hdc,point.x-R,point.y-R,
                point.x+R,point.y +R);
            break;
        case RIGHT:
            Ellipse(hdc,point.x-R,point.y-R,
                point.x+R,point.y +R);
            Ellipse(hdc,point.x+R,point.y-R,
                point.x+3*R,point.y+R);
            break;
        }
        frame++;                                               //准备绘制下一帧
        SelectObject(hdc,oldPen);
        DeleteObject(pen);
    return;
}
```

PacMan 与敌人都有各自的画法。PacMan 的绘制是一个由 v 字小开口的圆弧、圆、半圆构成动画，看起来是一个每时每刻都在活动的“大嘴”。它为 4 个方向和一个被抓住的状态。当它被抓住时，不绘制动画。每个方向都是由 3 种画面构成的 4 帧循环动画。

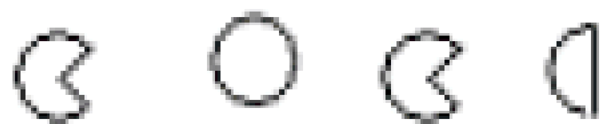
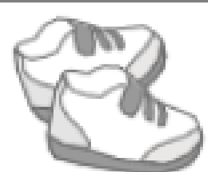


图 17.22　“大嘴”4 帧动画

PacMan 的绘制函数如下：

```
void PacMan::Draw( HDC& memDC)
{
    if(tw == OVER)
```






```
{

}
else if(frame%2 ==0)                                          //第 4 帧动画与第 2 帧动画
{
        int x1=0,x2=0,y1=0,y2=0;
        int offsetX =   DISTANCE/2+D_OFFSET;                  //弧弦交点
        int offsetY =   DISTANCE/2+D_OFFSET;                  //弧弦交点
switch(tw)
{
        case UP:
                x1 = point.x - offsetX;
                x2 = point.x + offsetX;
                y2 = y1 = point.y-offsetY;
                break;
        case DOWN:
                x1 = point.x + offsetX;
                x2 = point.x - offsetX;
                y2 = y1 = point.y+offsetY;
                break;
        case LEFT:
                x2 = x1 = point.x-offsetX;
                y1 = point.y + offsetY;
                y2 = point.y - offsetY;
                break;
        case RIGHT:
                x2 = x1 =point.x + offsetX;
                y1 = point.y - offsetY;
                y2 = point.y + offsetY;
                break;
}
        Arc(memDC,point.x-DISTANCE,point.y-DISTANCE,
        point.x+DISTANCE,point.y+DISTANCE,
        x1,y1,
        x2,y2);
        MoveToEx(memDC,x1,y1,NULL);
        LineTo(memDC,point.x,point.y);
        LineTo(memDC,x2,y2);
        }
else if(frame%3 ==0)
{
        Ellipse(memDC,point.x-DISTANCE,point.y-DISTANCE,
        point.x+DISTANCE,point.y+DISTANCE);
}
else {
        int x1=0,x2=0,y1=0,y2=0;
        switch(tw)
        {
```



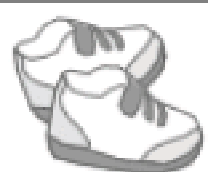


```
            case UP:
                x1 = point.x - DISTANCE;
                x2 = point.x + DISTANCE;
                y2 = y1 = point.y;
                break;
            case DOWN:
                x1 = point.x + DISTANCE;
                x2 = point.x - DISTANCE;
                y2 = y1 = point.y;
                break;
            case LEFT:
                x2 = x1 = point.x;
                y1 = point.y + DISTANCE;
                y2 = point.y - DISTANCE;
                break;
            case RIGHT:
                x2 = x1 =point.x ;
                y1 = point.y - DISTANCE;
                y2 = point.y + DISTANCE;
                break;
            }
            Arc(memDC,point.x-DISTANCE,point.y-DISTANCE,
            point.x+DISTANCE,point.y+DISTANCE,
            x1,y1,
            x2,y2);
            MoveToEx(memDC,x1,y1,NULL);
            LineTo(memDC,point.x,point.y);
            LineTo(memDC,x2,y2);
        }
        frame++;                                        //绘制下一帧
}
```

在 Enermy 的子类用不同颜色的画笔绘制轮廓，在它们的构造函数中将颜色成员变量初始化：

```
RedOne(int x,int y):Enermy(x,y)
{
    color = RGB(255,0,0);
}
BlueOne(int x,int y):RedOne(x,y)
{

    color = RGB(0,0,255);
}
YellowOne(int x,int y):RedOne(x,y)
{
    color = RGB(200,200,100);
}
```



17.3.6 窗口设计



Note

在 Visual Studio 中创建一个 Windows 窗口应用程序，在这个窗口框架中使用游戏中的各种类。在 pacman.cpp 中添加类所在的头文件和一些宏：

```
#include "pacman.h"
#include "GObject.h"
#define WLENTH 700
#define WHIGHT 740
#define STAGE_COUNT 3                                    //关卡数
```

在全局变量中声明物体子类的指针：

```
PacMan* p ;
GObject* e1;
GObject* e2 ;
GObject* e3 ;
GObject* e4 ;
```

在入口_tWinMain 函数代码最开始的地方，将它们初始化：

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPTSTR lpCmdLine,
                     int nCmdShow)
{
      UNREFERENCED_PARAMETER(hPrevInstance);
      UNREFERENCED_PARAMETER(lpCmdLine);
      // TODO: 在此放置代码
      int s_n = 0;                                       //进行到的关卡数
      p = new PacMan(P_ROW,P_ARRAY);
      e1 =new RedOne(E_ROW,E_ARRAY);
      e2 =new RedOne(E_ROW,E_ARRAY);
      e3 = new BlueOne(E_ROW,E_ARRAY);
      e4 = new YellowOne(E_ROW,E_ARRAY);
      GMap* MapArray[STAGE_COUNT] = {new Stage_1(),new Stage_2(),new Stage_3()};
      GObject::pStage =MapArray[s_n];                    //初始化为第一关地图
      Enermy::player = p;
```

上述代码还将地图中 3 个关卡的指针放入到一个数组中，物体类的地图指针指向了第一关。敌人追踪的玩家设定为 p。

在程序中使用了动态分配，需要使用堆内存回收。定义一个函数模板：

```
template<class T>
void Realese(T t)
{
if(t!=NULL)
```



```
        delete t;
}
```

在这个函数模板中传入指针变量，即可对它所指向的堆内存回收。



Note

绘图时会使用到当前窗口的设备上下文，那么首先应该获得这个窗口的句柄。这个窗口的句柄在代码中的函数 InitInstance 出现过：

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
        HWND hwnd;
         hInst = hInstance;                                    //将实例句柄存储在全局变量中
         hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
              0, 0, WLENTH, WHIGHT, NULL, NULL, hInstance, NULL);
        if (!hWnd)
     {
           return FALSE;
     }
…
}
```

但是函数并没有反馈给_tWinMain 窗口句柄。现在改动这个函数：

```
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow,HWND& hWnd)
{
   hInst = hInstance;                                          //将实例句柄存储在全局变量中

   hWnd = CreateWindow(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
               0, 0, WLENTH, WHIGHT, NULL, NULL, hInstance, NULL);
     if (!hWnd)
     {
         return FALSE;
     }
…
}
```

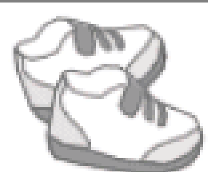
在_tWinMain 使用这个函数之前创建一个窗口句柄传递进来：

```
HWND hWnd;
if (!InitInstance (hInstance, nCmdShow,hWnd))
{
        return FALSE;
}
```

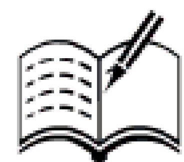
这样在主函数中就获得了窗口句柄。为何要在主函数中绘图，而不在窗口过程函数中处理 WM_PAINT 时绘制它呢？

在程序中使用的消息循环为如下形式：

```
while (GetMessage(&msg, NULL, 0, 0))
{
```



```
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
}
```



Note

GetMessge 可以获得消息队列中的消息。当没有消息传递时，将会“冻结”窗口。而 WM_PAIT 消息是在水平移动或者窗口大小发生改变时会重新绘制窗口的图像。

我们要进行的是一个实时游戏，这两种情况都不符合我们的需求。API 中还有另外一种获得消息队列的函数 PeekMessage，它接收的参数比 GetMessage 多出一项：

```
PeekMessage(&msg, NULL, 0, 0,PM_REMOVE)
```

最后一个参数代表它处理消息的形式，设置为 PM_REMOVE 表示处理消息后将在队列中消去它。

PeekMessage 不会在队列无消息时冻结窗口程序。在新的消息循环中应该考虑程序在何种情况下结束：

游戏失败、闯过所有关卡或者关闭窗口。

在销毁窗口时，对应着窗口过程函数的 WM_DESTROY 的消息处理。将它更改为：

```
case WM_DESTROY:
    PostQuitMessage(0);
    ::exit(0);
    break;
```

使用 exit 可以退出应用程序，这样就不会出现窗口销毁而当前的应用程序仍在运行的情况。

在主函数的消息循环中循环的条件应该为游戏失败或者闯过所有关卡。这时需要获得“大嘴”的方向状态。在“大嘴”中定义获得方向的函数：

```
TWARDS PacMan::GetTw()
{
return tw;
}
```

当前的循环更改为：

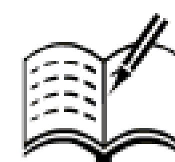
```
while(p->GetTw()!=OVER&&s_n<3)
{
if(PeekMessage(&msg, NULL, 0, 0,PM_REMOVE))
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
}
```

在游戏中，使用键盘控制“大嘴”。这样就需要能够改变“大嘴”当前的方向指令，定义以



下成员函数：

```
void PacMan::SetTwCommand(TWARDS command)
{
twCommand = command;
}
```



获得键盘状态的 API：

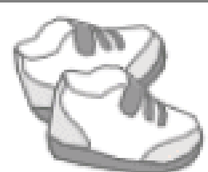
```
GetAsyncKeyState(int key)
```

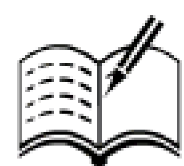
key 代表的是键盘各个键位的数字码，Windows 定义了宏来代替使用数字代码的形式传入此函数。

UP、DOWN、LEFT、RIGHT 分别代表上、下、左、右 4 个方向键。

在循环中添加游戏内容：

```
while(p->GetTw()!=OVER&&s_n<3)
{
	if(p->Win())
	{
		HDC hdc = GetDC(hWnd);
		s_n++;
		ResetGObjects();
		if(s_n <3)
		{
			MessageBoxA(hWnd,"恭喜您过关","吃豆子提示",MB_OK);
			GObject::pStage = MapArray[s_n];
			RECT screenRect;
			screenRect.top = 0;
			screenRect.left = 0;
			screenRect.right = WLENTH;
			screenRect.bottom = WHIGHT;
			::FillRect(hdc,&screenRect,CreateSolidBrush(RGB(255,255,255)));
			GObject::pStage->DrawMap(hdc);
		}
		continue;
	}
	if(PeekMessage(&msg, NULL, 0, 0,PM_REMOVE))
	{
		TranslateMessage(&msg);
		DispatchMessage(&msg);
	}
	if(GetAsyncKeyState(VK_DOWN)&0x8000)
	{
		p->SetTwCommand(DOWN);
	}
	if(GetAsyncKeyState(VK_LEFT)&0x8000)
	{
		p->SetTwCommand(LEFT);
```





Note

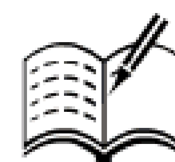
```
	}
	if(GetAsyncKeyState(VK_RIGHT)&0x8000)
	{
		p->SetTwCommand(RIGHT);
	}
	if(GetAsyncKeyState(VK_UP)&0x8000)
	{
		p->SetTwCommand(UP);
	}
	else
	{
		if(GetTickCount()-t>58)
		{
			HDC hdc = GetDC(hWnd);
			e1->action();
			e2->action();
			e3->action();
			e4->action();
			p->action();
			GObject::pStage->DrawPeas(hdc);
			e1->DrawBlank(hdc);
			e2->DrawBlank(hdc);
			e3->DrawBlank(hdc);
			e4->DrawBlank(hdc);
			p->DrawBlank(hdc);
			e1->Draw(hdc);
			e2->Draw(hdc);
			e3->Draw(hdc);
			e4->Draw(hdc);
			p->Draw(hdc);
			DeleteDC(hdc);
			t = GetTickCount();
		}
	}
}
```

在代码中出现了 DrawBlank 这个 GObject 的成员函数。现在来讲解它的实现与作用：

```
void GObject::DrawBlank(HDC& hdc)
{
	RECT rect;
	rect.top = mY-RD;
	rect.left = mX-RD;
	rect.right = mX+RD;
	rect.bottom = mY+RD;
	FillRect(hdc,&rect,::CreateSolidBrush(RGB(255,255,255)));
}
```

程序绘制图像时，不会将上一帧绘制的图形自动擦去。每次绘图之前，将上一次绘图区域所在的矩形用背景底色覆盖掉，然后再绘制新图形使物体对象看上去真如同移动一样。





成员变量 mY、mX 记录的就是一次物体中心所在的位置。在碰撞检测中，移动物体坐标前更新它们的数据方向：

```
…
mX = point.x;
mY = point.y;
int MAX =pStage-> LD*MAPLENTH+pStage->LD/2;
int MIN = pStage->LD/2;
switch(tw)                                          //判断行进的方向
…
```

消息循环中还存在着这样一个段代码：

```
if(GetTickCount()-t>58)
{
      …;
}
```

GetTickCount 函数获得的是从开机到当前时刻机器运行的毫秒数。在消息循环外使用一个无符号长整型变量 t 储存游戏计时：

```
DWORD t =0;
```

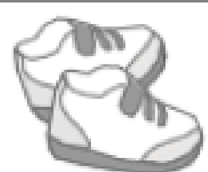
在循环返回前更新它：

```
t = GetTickCount();
```

每 58 毫秒游戏的数据和画面会更新一次。物体的速度相当于 speed 成员变量除以 58 毫秒 ，以“闪烁”的方式平移到相应的 speed 数目的像素。

在玩家获得某一关的胜利时，所有物体位置会“还原”，地图使用背景色刷新之后再绘制新关卡的地图：

```
if(p->Win())
      {
            HDC hdc = GetDC(hWnd);
            s_n++;
            ResetGObjects();
            if(s_n <3)
            {
                  MessageBoxA(hWnd,"恭喜您过关","吃豆子提示",MB_OK);
                  GObject::pStage = MapArray[s_n];
                  RECT screenRect;
                  screenRect.top = 0;
                  screenRect.left = 0;
                  screenRect.right = WLENTH;
                  screenRect.bottom = WHIGHT;
                  ::FillRect(hdc,&screenRect,CreateSolidBrush(RGB(255,255,255)));
                  GObject::pStage->DrawMap(hdc);
            }
```



```
            continue;
    }
```



Note

当玩家获得某张地图的胜利分为两种情况：进入下一关或者游戏结束。

程序中声明并实现 ResetGObjects 函数：

```
void ResetGObjects()
{
    p->SetPosition(P_ROW,P_ARRAY);
    e1->SetPosition(E_ROW,E_ARRAY);
    e2->SetPosition(E_ROW,E_ARRAY);
    e3->SetPosition(E_ROW,E_ARRAY);
    e4->SetPosition(E_ROW,E_ARRAY);
}
```

SetPosition 函数是 GObject 类声明的设置自身中心位置的函数：

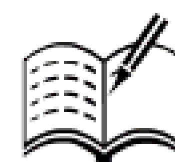
```
void GObject::SetPosition(int Row,int Array)
{
    dRow = Row;
    dArray = Array;
    this->point.y = dRow*pStage->LD+pStage->LD/2;
    this->point.x = dArray*pStage->LD+pStage->LD/2;
}
```

MessageBox 函数是 Windows API 中弹出对话框的函数，通过参数设定可以设置它的类型和文字表达。在这里它的作用是提示玩家顺利闯过关卡。

循环执行之后，程序即将结束。在此之前再次使用 MessageBox 提示玩家胜利或者失败：

```
Realese(e1);
Realese(e2);
Realese(e3);
Realese(e4);
for(int i = 0;i<STAGE_COUNT;i++)
{
    Realese(MapArray[i]);
}
if(p->GetTw()==OVER)
{
    MessageBoxA(hWnd,"出师未捷","吃豆子提示",MB_OK);
}
else
{
    MessageBoxA(hWnd,"恭喜您赢得了胜利","吃豆子提示",MB_OK);
}
Realese(p);
return (int) msg.wParam;
```





17.3.7　小结

本节设计了一个吃豆子游戏，目的是让读者理解类的多态与继承的使用方法。Windows API 很繁重，学习它需要大量时间和实践，如何驾驭它并不是本章的重点。地图类与物体类使用了 C++类继承的特性，子类的共性应该放入父类中，性质相似而实现不同的函数则应该由虚函数定义。

第18章

人事考勤管理系统
（Visual Studio 2010和SQL Server 2008实现）

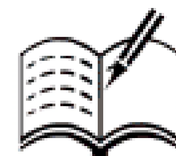
（视频讲解：1小时30分钟）

一个小的企业由于员工不多，对于员工的出勤管理可能不是一个问题。但如果随着企业的不断壮大，员工不断增加，那么员工的出勤管理就会成为非常大的问题。所以人事考勤管理系统就是为了解决这个问题而产生的。有了人事考勤管理系统就可以轻易地掌握每个员工的出勤状况。

本章能够掌握的主要知识点（已掌握的在方框中打勾）

- [ ] 了解如何使用ADO连接数据库
- [ ] 学习如何利用ADO封装类进行数据操作
- [ ] 了解数据库与程序间日期类型数据的操作
- [ ] 学习如何使用SQL查询语句进行数据表的汇总查询





18.1　开发背景

XX 公司随着业务的不断发展，公司的员工数量不断增加，人事考勤方面的管理已成为公司管理中的重要部分。传统的人事考勤制度已不能有效地管理员工的出勤状况，所以人事考勤系统必然就成为人事考勤管理的有效工具。

18.2　需求分析

在使用人事考勤管理系统时，用户需要输入用户名和密码进入系统，对其中的部门、员工的基本信息进行维护和管理。在考勤管理模块中录入员工当天的考勤信息，同时可对年、月、员工进行查询。还可通过考勤汇总查询对员工某月的考勤记录进行汇总，计算出员工月工作天数、早退、迟到的天数等。通过对人事考勤管理过程的研究和分析，要求本系统应该具有以下功能：

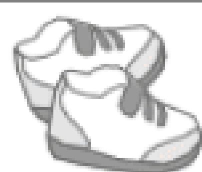
☑　用户登录。
☑　部门信息录入。
☑　人员信息管理。
☑　考勤信息录入。
☑　考勤信息汇总。

18.3　系统设计

18.3.1　系统目标

人事考勤管理系统以实现员工日常出勤信息管理为设计目标，加以强大的数据库管理功能，可以方便地对考勤信息进行管理，大大地提高人事部门的日常工作效率。本系统在设计时应该满足以下几点：

☑　采用人机对话的操作方式，信息查询灵活、方便、快捷、准确、数据存储安全可靠。
☑　对考勤信息的操作简单，可以方便地进行添加、修改和删除。
☑　可以录入员工信息、部门信息。
☑　对员工的考勤信息可按月进行汇总计算。
☑　对用户输入的数据，系统进行严格的数据检验，尽可能排除人为的错误。
☑　系统最大限度地实现了易维护性和易操作性。
☑　系统运行稳定、安全可靠。





Note

18.3.2 系统功能结构

人事考勤管理系统的功能结构图如图 18.1 所示。

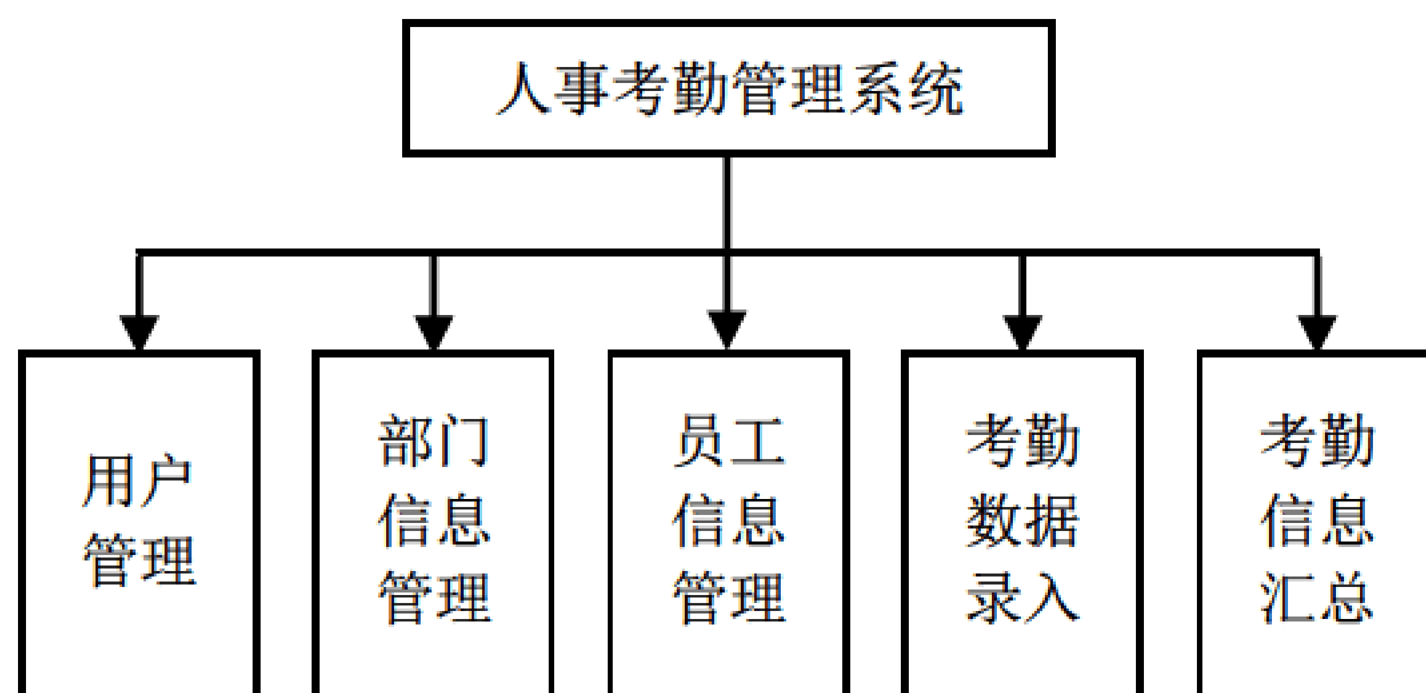


图 18.1 系统功能结构图

18.3.3 系统预览

人事考勤管理系统由多个功能模块组成，下面仅列出几个典型的功能模块，其他模块请参见光盘中的源程序。

人事考勤部门信息管理如图 18.2 所示，该模块用于管理各部门之间的结构信息；人事考勤员工管理如图 18.3 所示，该模块用于维护员工的基本信息。



图 18.2 人事考勤部门信息管理

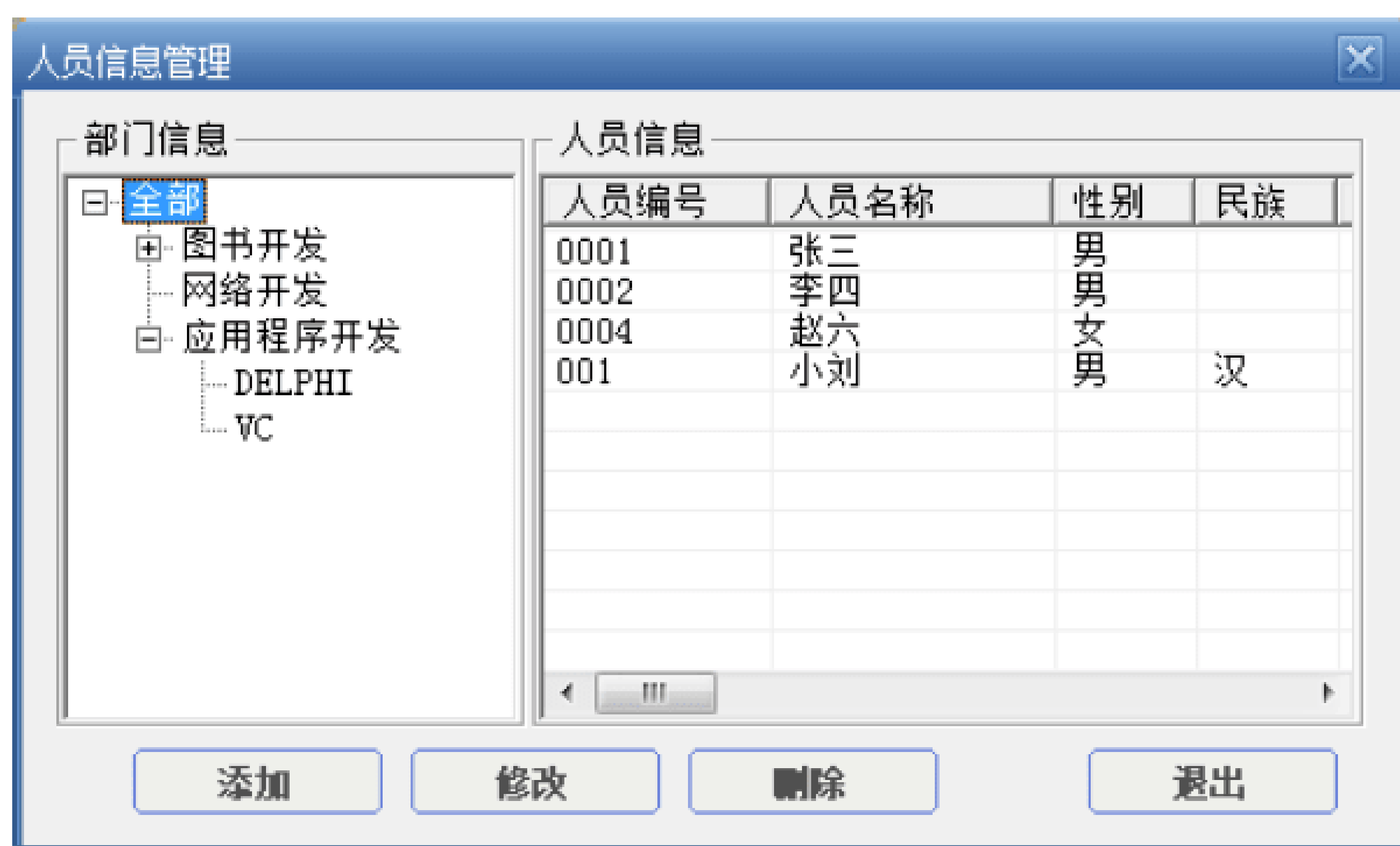


图 18.3 人事考勤信息查询

人事考勤管理模块如图 18.4 所示，该模块用于记录人事考勤的信息情况；人事考勤汇总模块如图 18.5 所示，该模块用于对员工的考勤信息进行汇总统计。

18.3.4 业务流程图

人事考勤管理系统的业务流程图如图 18.6 所示。





考勤管理

☑ 显示全部　年：2013　月：1　员工：(全部)

人员姓名	上班时间	下班时间	上班考勤时间	下班考勤时间	请假类别
李四	8:00:00	17:00:00	8:00:00	17:00:00	无
小刘	8:00:00	17:30:00	8:00:00	15:30:00	无
张三	8:00:00	17:00:00	10:10:00	16:50:00	无

添加　修改　删除　退出

图 18.4　人事考勤管理模块

考勤汇总查询

年：2009　月：5　员工：张三　退出

人员姓名	工作总天数	迟到总天数	早退总天数	病假总天数	事假总
张三	.83	.27	.02	0	0

图 18.5　人事考勤汇总模块

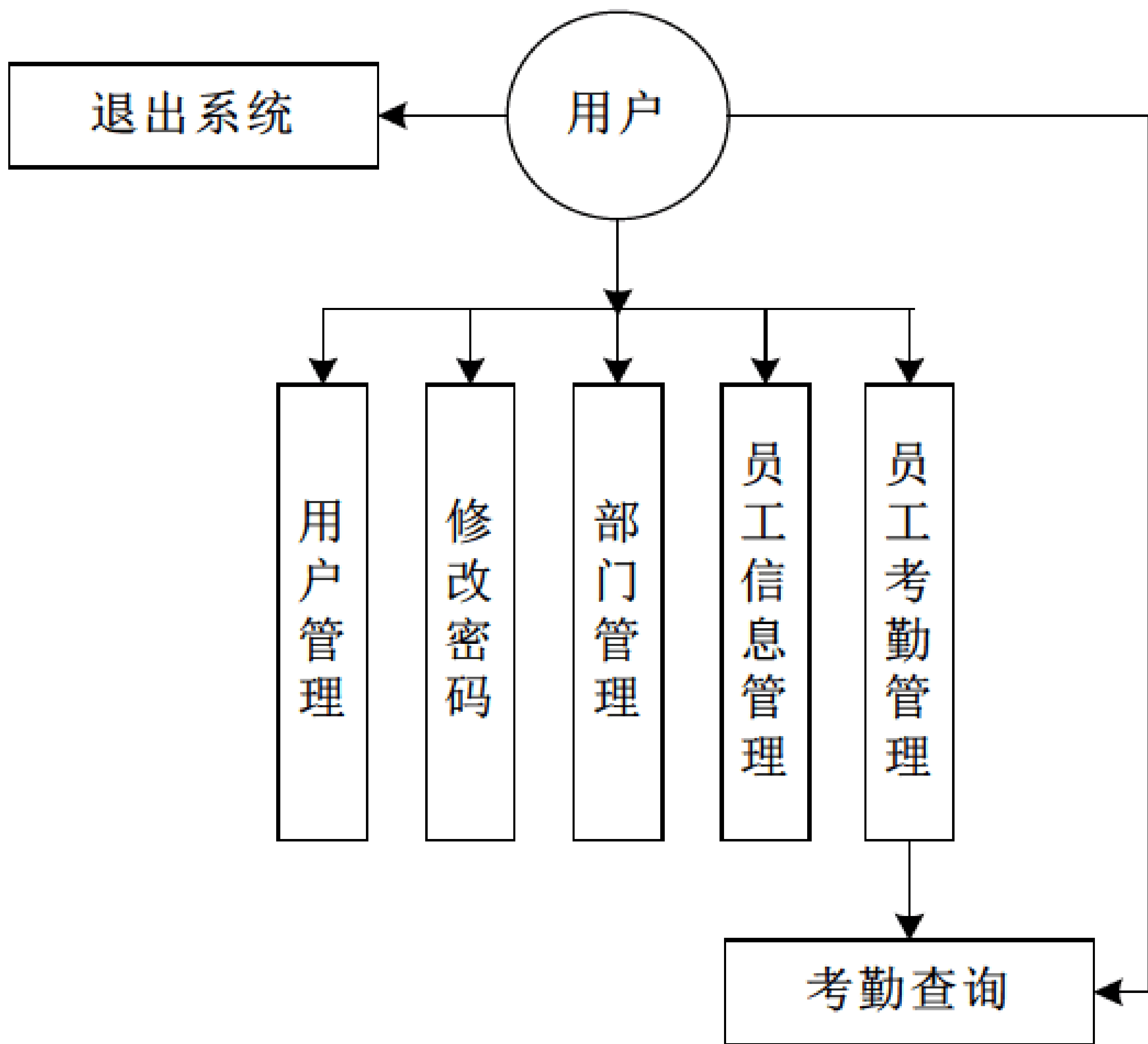


图 18.6　人事考勤管理系统的业务流程图

18.3.5　数据库设计

1．数据库分析

在人事考勤管理系统使用了 Microsoft SQL Server 2008 数据库来满足系统的要求，数据库名称为 tb_person，在数据库中创建 4 张表用于存储各种不同的信息，如图 18.7 所示。

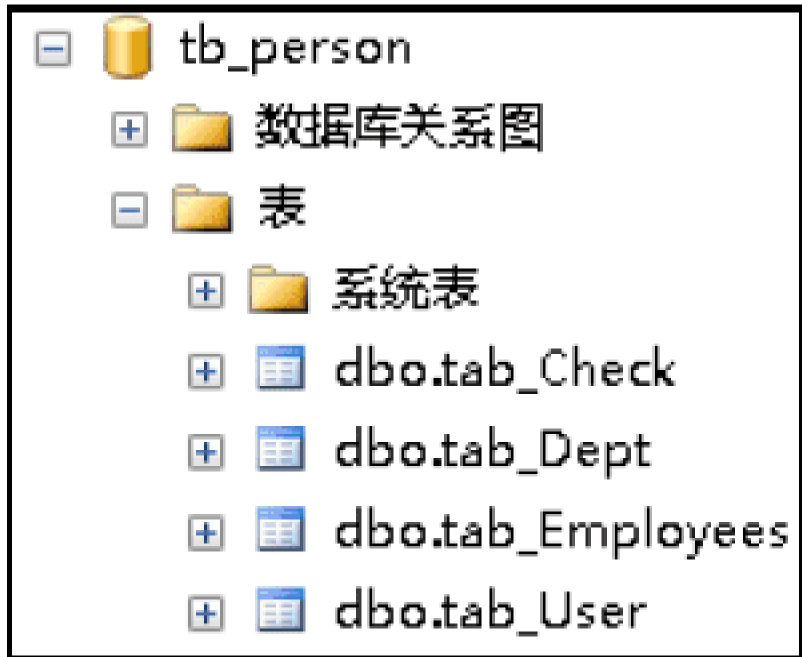


图 18.7　数据库中的表

2．数据库概念设计

根据前面介绍的需求分析和系统设计规划出本系统中使用的数据库实体对象，分别为管理员实体、部门实体、员工实体和考勤实体等。下面将给出各实体的 E-R 图。

管理员实体包括编号、管理员姓名、密码信息。管理员实




Note

体 E-R 图如图 18.8 所示。

部门实体包括部门编号、部门名称、备注信息和上级部门编号。部门实体 E-R 图如图 18.9 所示。

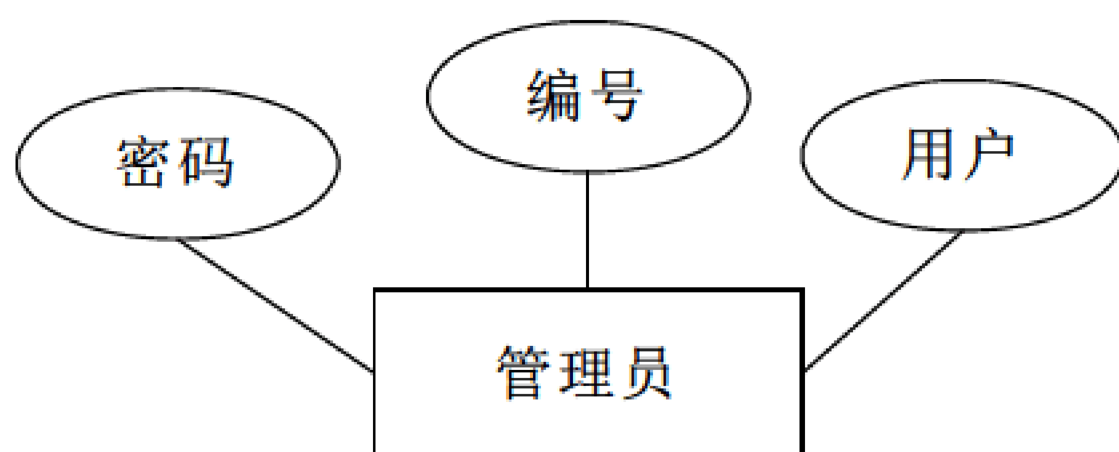


图 18.8　管理员实体 E-R 图

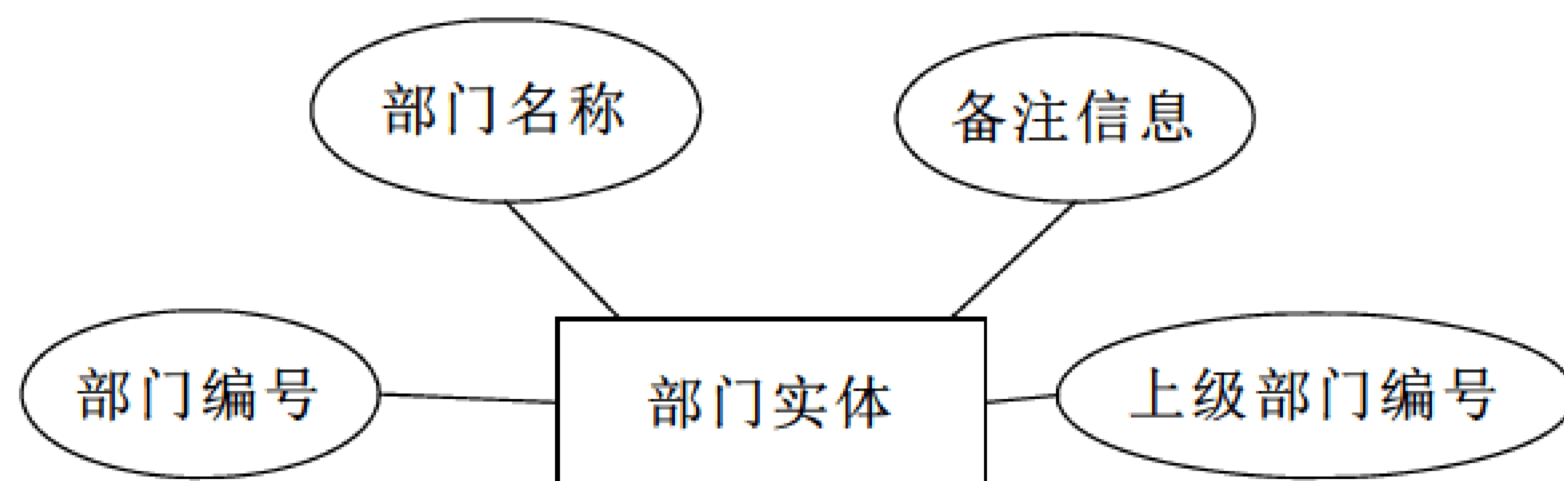


图 18.9　部门实体 E-R 图

员工实体包括自动编号、员工编号、员工姓名、照片、性别、民族和生日等信息。员工实体 E-R 图如图 18.10 所示。

考勤实体包括人员姓名、考勤日期、上班时间、下班时间、上班考勤时间和下班考勤时间等信息。考勤实体 E-R 图如图 18.11 所示。

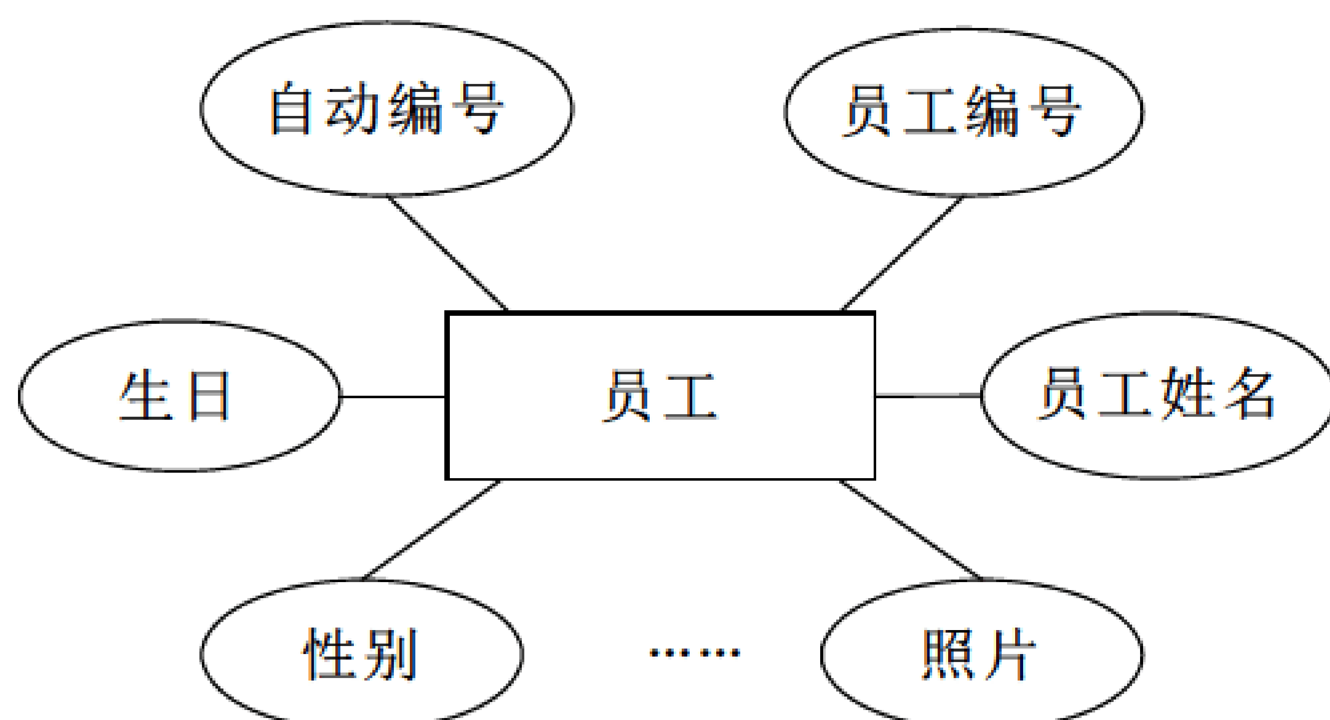


图 18.10　员工实体 E-R 图

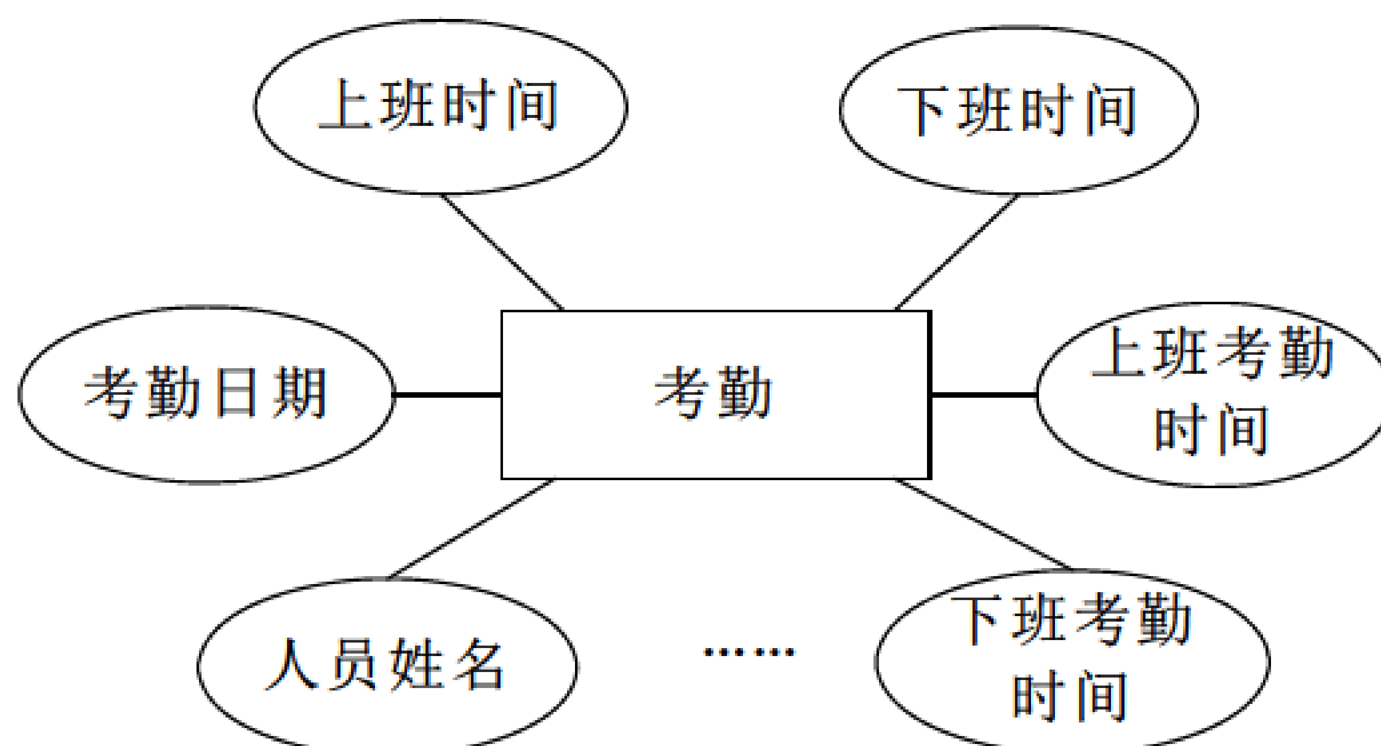


图 18.11　考勤实体 E-R 图

3．数据库逻辑结构设计

本例使用的是 SQL Server 2008 数据库，根据实体 E-R 图创建各数据表。下面给出人事考勤管理系统数据库中主要表的表结构。

tab_User（管理员信息表）用于保存管理员的信息，如图 18.12 所示。

tab_Dept（部门信息表）用于记录部门的信息情况，如图 18.13 所示。

MRWXK\MRWXK.tb_p...n - dbo.tab_User

列名	数据类型	允许 Null 值
ID	int	☐
UserName	varchar(50)	☐
PassWord	varchar(50)	☐

图 18.12　tab_User（管理员信息表）

MRWXK\MRWXK.tb_p...n - dbo.tab_Dept

列名	数据类型	允许 Null 值
ID	int	☐
DeptName	varchar(50)	☐
Memo	varchar(50)	☑
PID	int	☐

图 18.13　tab_Dept（部门信息表）

tab_Employees（员工信息表）用来保存公司员工信息，如图 18.14 所示。

tab_Check（考勤信息表）记录员工每天的考勤信息，如图 18.15 所示。





MRWXK\MRWXK.tb_...bo.tab_Employees

列名	数据类型	允许 Null 值
AutoID	int	☐
Emp_Id	varchar(50)	☐
Emp_NAME	varchar(50)	☐
Photo	image	☑
Sex	char(2)	☑
Nationality	varchar(40)	☑
Birth	varchar(20)	☑
Political_Party	varchar(40)	☑
Culture_Level	varchar(40)	☑
Marital_Condition	varchar(20)	☑
Family_Place	varchar(60)	☑
Id_Card	varchar(20)	☑
Office_phone	varchar(30)	☑
Mobile	varchar(30)	☑
Files_Keep_Org	varchar(100)	☑
Hukou	varchar(100)	☑
HireDate	varchar(20)	☑
Dept	int	☑
Duty	varchar(40)	☑
Memo	varchar(200)	☑

图 18.14　tab_Employees（员工信息表）

MRWXK\MRWXK.tb_... - dbo.tab_Check

列名	数据类型	允许 Null 值
autoid	int	☐
name	varchar(50)	☐
checkdate	datetime	☐
ondutytime	datetime	☐
offdutytime	datetime	☐
ontime	datetime	☐
offtime	datetime	☐
leave	varchar(50)	☑
onleave	datetime	☐
offleave	datetime	☐
latetime	datetime	☐
leaveearly	datetime	☐
memo	varchar(200)	☑

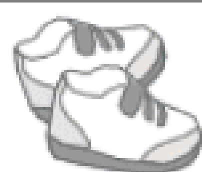
图 18.15　tab_Check（考勤信息表）

18.4　公共模块设计

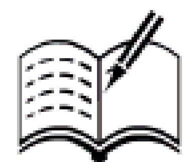
本系统是使用 ADO 连接数据库的，为了能更方便地在程序中使用 ADO 建立数据库连接与数据表的操作，就在公共类中对系统中所使用的 ADO 操作进行了封装。在该系统中建立了 ADO 的两个公共类 CADOConnection 和 CADODataSet，这两个类定义在 ADO.h 头文件中，实现在 ADO.cpp 文件中。

CADOConnection 类是用来连接数据库的，实现了对_Connection 接口的封装。CADOConnection 类在头文件中的定义如下：

```
//载入 msado15dll，这样在工程中就不必再载入了
#import "msado15.dll"    no_namespace rename("EOF","adoEOF")
class CADOConnection
{
private:
        static void InitADO();                              //初始化 ADO
        static void UnInitADO();
protected:
        _ConnectionPtr m_Connection;                        //接口指针
public:
        BOOL IsOpen();                                      //判断是否与数据库连接
        _ConnectionPtr GetConnection();                     //获取连接接口
```



```
        CString GetSQLConStr(CString IP,CString DBName);          //获取 SQL 连接字符串
        BOOL Open(CString ConStr);                                 //建立数据库连接
        CADOConnection();
        virtual ~CADOConnection();
};
CADOConnection * GetConnection();                                  //获取全局连接类的函数
```



Note

定义两个全局变量 ConCount 和 g_Connection，ConCount 变量是一个整型变量，用来记录在工程中所创建的 CADOConnection 类的实例个数。在构造方法中当此变量为 0 时调用 CoInitialize 函数实现 OLE 的初始化。在析构方法中当此变量为 0 时调用 CoUninitialize 方法取消 OLE 的初始化。

```
int ConCount = 0;
CADOConnection g_Connection;                                       //全局数据库连接对象
```

GetConnection 函数是一个全局函数，用于返回全局数据库连接对象的指针。代码如下：

```
CADOConnection * GetConnection()
{
        return &g_Connection;
}
```

CADOConnection 方法是构造函数，用于初始化 OLE 和创建_Connection 接口的指针实例。代码如下：

```
CADOConnection::CADOConnection()
{
        InitADO();
        m_Connection.CreateInstance("ADODB.Connection");
}
```

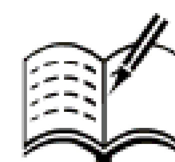
～CADOConnection 方法是析构函数，用于取消 OLE 的初始化和释放_Connection 接口指针。代码如下：

```
CADOConnection::~CADOConnection()
{
        if (IsOpen())
                m_Connection->Close();
        m_Connection = NULL;
        UnInitADO();
}
```

InitADO 方法是一个静态方法，用于初始化 OLE。代码如下：

```
void CADOConnection::InitADO()
{
        if (ConCount++ == 0)
                CoInitialize(NULL);
}
```





UnInitADO 方法是一个静态方法，用于取消 OLE 的初始化。代码如下：

```
void CADOConnection::UnInitADO()
{
	if (--ConCount == 0)
		CoUninitialize();
}
```

Open 方法通过指定的数据库连接字符串与 SQL 数据库建立连接。代码如下：

```
BOOL CADOConnection::Open(CString ConStr)
{
	if (IsOpen())
		m_Connection->Close();
	m_Connection->Open((_bstr_t)ConStr,"","",adModeUnknown);
	return IsOpen();
}
```

GetSQLConStr 方法用来生成与数据库连接所需要的连接字符串。代码如下：

```
CString CADOConnection::GetSQLConStr(CString IP, CString DBName)
{
	CString Str;
	Str.Format("Provider=SQLOLEDB.1;Persist Security Info=False;\
		User ID=sa;Initial Catalog=%s;Data Source=%s",DBName,IP);
	return Str;
}
```

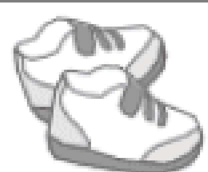
GetConnection 方法用于返回_Connection 接口指针。代码如下：

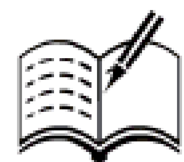
```
_ConnectionPtr CADOConnection::GetConnection()
{
	return m_Connection;
}
```

IsOpen 方法用来判断当前数据库连接对象与数据库的连接状态。代码如下：

```
BOOL CADOConnection::IsOpen()
{
	long State;
	m_Connection->get_State(&State);
	if (State == adStateOpen)
		return true;
	return false;
}
```

CADODataSet 类是用来存储数据的数据集类，该类实现了_Recordset 接口的实例。该类在头文件中的定义如下：






```
class CADODataSet
{
protected:
	_RecordsetPtr m_DataSet;                                    //数据集接口指针
	CADOConnection *m_Connection;                               //数据库连接类对象
public:
	void Delete();                                              //记录删除
	int GetRecordNo();                                          //获取记录集行号
	void move(int nIndex);                                      //移动记录指针
	void Save();                                                //保存对记录集的修改
	void SetFieldValue(CString FieldName,_variant_t Value);     //设置字段的值
	void AddNew();                                              //添加新记录
	BOOL Next();                                                //记录集指针指向下一条记录
	FieldsPtr GetFields();                                      //获取记录集字段集合
	int GetRecordCount();                                       //获取记录集中记录数量
	void SetConnection(CADOConnection *pCon);                   //设置记录集的数据库连接对象
	BOOL Open(CString SQLStr);                                  //打开记录集

	CADODataSet();
	virtual ~CADODataSet();
private:
	BOOL IsOpen();                                              //判断记录集是否打开
};
```

CADODataSet 方法为记录集实现类的构造方法，在该方法中实现记录集接口对象的创建。代码如下：

```
CADODataSet::CADODataSet()
{
	m_DataSet.CreateInstance("ADODB.Recordset");
}
```

～CADODataSet 类为记录集实现类的析构方法，在该方法中实现记录集的关闭与接口的释放。代码如下：

```
CADODataSet::~CADODataSet()
{
	if (IsOpen())
		m_DataSet->Close();
	m_DataSet = NULL;
	m_Connection = NULL;
}
```

SetConnection 方法用来设置记录集所连接的数据库连接类的对象。代码如下：

```
void CADODataSet::SetConnection(CADOConnection *pCon)
{
	m_Connection = pCon;
}
```





GetRecordCount 方法用来获取记录集中数据的数量。实现代码如下：

```
int CADODataSet::GetRecordCount()
{
	if (IsOpen())
		return m_DataSet->GetRecordCount();
	else
		return 0;
}
```

Open 方法通过 SQL 查询语句打开数据集。实现代码如下：

```
BOOL CADODataSet::Open(CString SQLStr)
{
	if (IsOpen())
		m_DataSet->Close();
	m_DataSet->Open(_bstr_t(SQLStr),
			_variant_t((IDispatch*)g_Connection.GetConnection(), true),
			adOpenKeyset, adLockUnspecified, adCmdText);
	return IsOpen();
}
```

IsOpen 方法用来判断数据集是否处于打开状态。实现代码如下：

```
BOOL CADODataSet::IsOpen()
{
	long State;
	m_DataSet->get_State(&State);
	if (State == adStateOpen)
		return true;

	return false;
}
```

GetFields 方法用来获取记录集中字段的集合。实现代码如下：

```
FieldsPtr CADODataSet::GetFields()
{
	return m_DataSet->GetFields();
}
```

Next 方法将记录集指针下移一位。实现代码如下：

```
BOOL CADODataSet::Next()
{
	if (m_DataSet->adoEOF)
		return false;
	m_DataSet->MoveNext();
	return true;
}
```





Note

AddNew 方法用于向记录集中添加一个新行。实现代码如下：

```
void CADODataSet::AddNew()
{
	m_DataSet->AddNew();
}
```

SetFieldValue 方法用来向记录集中指定的字段赋值。实现代码如下：

```
void CADODataSet::SetFieldValue(CString FieldName, _variant_t Value)
{
	m_DataSet->PutCollect((_bstr_t)FieldName,Value);
}
```

Save 方法用来保存对记录集中所做的任何数据更改。实现代码如下：

```
void CADODataSet::Save()
{
	m_DataSet->Update();
}
```

Move 方法将记录集的当前指针移动到指定的索引位置。实现代码如下：

```
void CADODataSet::move(int nIndex)
{
	m_DataSet->MoveFirst();
	m_DataSet->Move(nIndex);
}
```

GetRecordNo 方法用来获取记录集中的当前行号。实现代码如下：

```
int CADODataSet::GetRecordNo()
{
	return m_DataSet->AbsolutePosition;
}
```

Delete 方法用来删除记录集中的当前行。实现代码如下：

```
void CADODataSet::Delete()
{
	m_DataSet->Delete(adAffectCurrent);
}
```

18.5 主窗体设计

人事考勤管理系统主窗口由菜单和客户区域组成，其中，客户区域显示了一幅位图，主窗体





效果如图 18.16 所示。



图 18.16 人事考勤管理系统的主窗体

主窗体设计步骤如下：

（1）启动 Visual Studio 2010，选择"文件"/"新建"命令，打开"新建项目"对话框。在"新建项目"对话框左方的列表视图中选择 Visual C++下的 MFC 应用程序，在"名称"文本框中输入解决方案名称，在"位置"文本框中设置解决方案保存的路径，如图 18.17 所示。

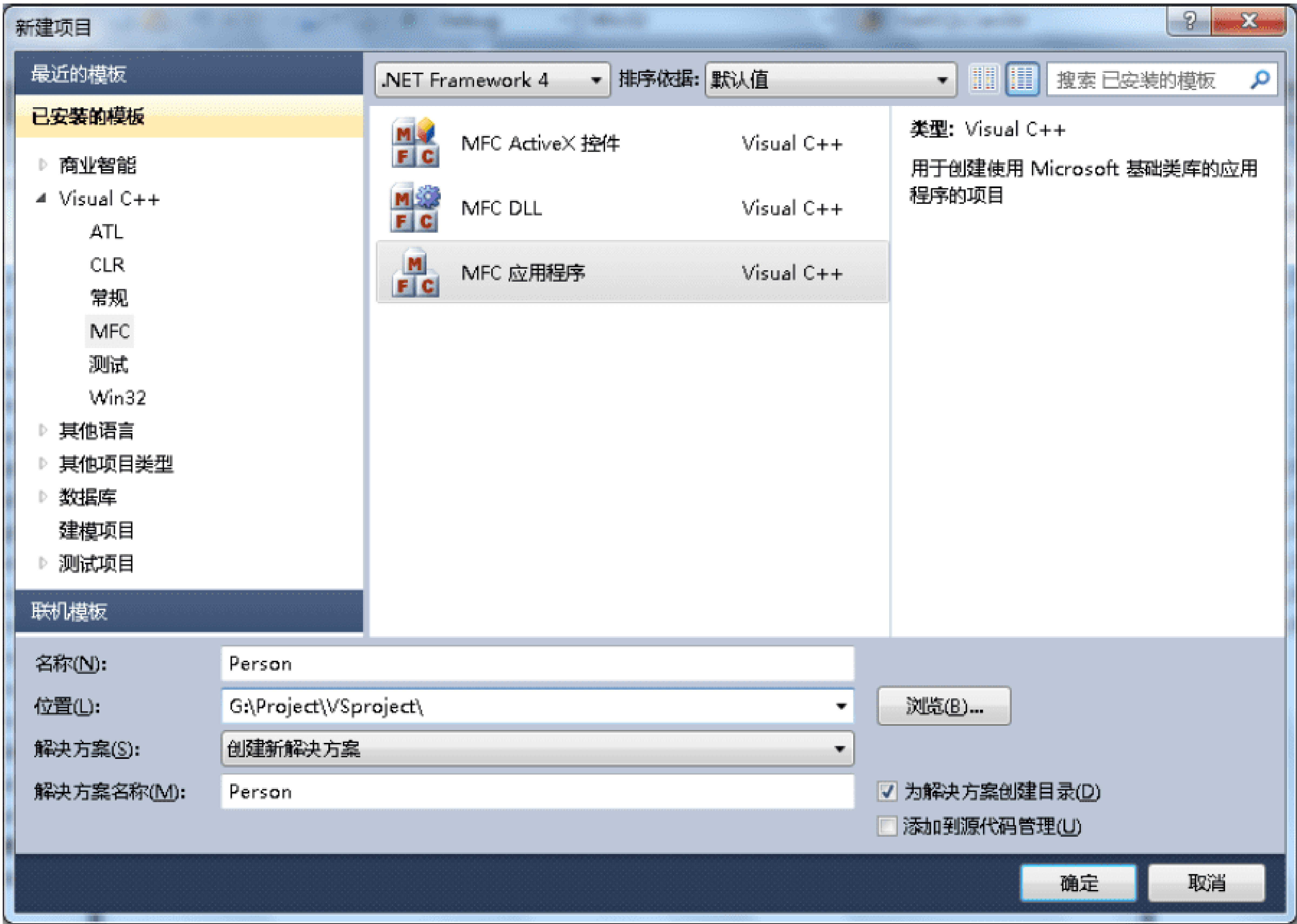
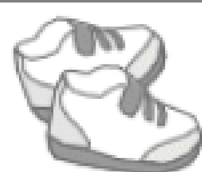


图 18.17 "新建项目"对话框





Note

（2）单击“确定”按钮进入“MFC 应用程序向导-Person”对话框，选中“基于对话框”单选按钮，如图 18.18 所示。



图 18.18　“MFC 应用程序向导-Person”对话框

（3）单击“完成”按钮完成工程的创建。

（4）在工作区窗口的“资源视图”中右击一个节点，在弹出的快捷菜单中选择“添加资源”命令，弹出“添加资源”对话框，如图 18.19 所示。



图 18.19　“添加资源”对话框

（5）选择 Menu 选项，单击“新建”按钮创建菜单资源，在“资源视图”中双击新创建的菜单资源，编辑菜单资源，菜单资源代码如下：

```
IDR_MAINMENU MENU DISCARDABLE
BEGIN
    POPUP "系统设置"                              //第一个菜单
```






```
    BEGIN                                         // “系统设置”菜单下的子菜单（菜单名称和 ID 号）
        MENUITEM "用户管理",                     ID_MENUUSER
        MENUITEM "修改密码",                     ID_MENUPASSWORD
        MENUITEM SEPARATOR
        MENUITEM "系统退出",                     ID_MENUEXIT
    END
    POPUP "基本信息管理"                          //第二个菜单
    BEGIN                                         // “基本信息管理”菜单下的子菜单
        MENUITEM "部门管理",                     ID_MENUDEPT
        MENUITEM "人员信息管理",                 ID_MENUPERSON
    END
    POPUP "员工考勤管理"                          //第三个菜单
    BEGIN                                         // “员工考勤管理”菜单下的子菜单
        MENUITEM "考勤管理",                     ID_MENUCHECK
        MENUITEM "考勤汇总查询",                 ID_MENUCHECKSUM
    END
END
```

18.6　用户登录模块设计

18.6.1　用户登录模块概述

用户登录模块是所有管理系统所应具备的基础模块之一，该模块实现了用户使用系统的检验工作，使没有权限的用户不能使用该系统，增加了系统的安全性。用户登录界面如图 18.20 所示。



图 18.20　用户登录模块

18.6.2　用户登录技术分析

用户登录窗体是整个系统中创建并显示的第一个窗体，所以该窗体应在主窗体创建前创建并显示。在登录窗体创建的同时应该创建数据库连接。这些操作都应在应用程序类的初始化方法中实现，该方法名为 InitInstance。代码如下：

```
BOOL CPersonApp::InitInstance()
{
    AfxEnableControlContainer();
#ifdef _AFXDLL
    Enable3dControls();
#else
    Enable3dControlsStatic();
#endif
    LoadSkin();
```





Note

```
	//创建全局数据库连接，将“1213.0.0.1”更换成数据库服务器名称
	BOOL bCon = GetConnection()->Open(GetConnection()->GetSQLConStr("1213.0.0.1","tb_person"));
	CLoginDialog logindlg;                     //定义登录窗体对象
	if (logindlg.DoModal() != IDOK)            //显示登录窗体
		return false;
	CPersonDlg dlg;                            //定义应用程序主窗体
	m_pMainWnd = &dlg;
	int nResponse = dlg.DoModal();             //显示主窗体
	if (nResponse == IDOK)
	{
	}
	else if (nResponse == IDCANCEL)
	{
	}
	return FALSE;
}
```

18.6.3　用户登录实现过程

本模块使用的数据表：Tab_User

（1）创建一个对话框，打开对话框属性窗口，将对话框的 ID 改为 IDD_DLGLOGIN，将对话框标题改为“登录”。

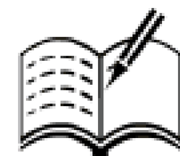
（2）向对话框中添加两个静态文本控件、一个编辑框控件、一个列表框控件和两个按钮控件。分别设置两个静态文本控件的 Caption 属性为“用户名：”和“密码：”，设置编辑框控件的类型为 password。分别设置两个按钮的 Caption 属性为“确定”和“取消”。

（3）在窗体的初始化方法中创建用户表的数据集，并将用户名添加到列表控件中。代码如下：

```
BOOL CLoginDialog::OnInitDialog()
{
	CDialog::OnInitDialog();
	m_DataSet.SetConnection(GetConnection());              //设置数据集连接的数据库连接
	                                                       //对象
	m_DataSet.Open("Select * From Tab_User");              //打开用户表
	int count = m_DataSet.GetRecordCount();                //获取用户数量
	for (int i = 0; i< count;i++)
	{                                                      //将用户名添加到列表控件中
		m_UserList.AddString((_bstr_t)m_DataSet.GetFields()->Item[L"UserName"]->Value);
		m_DataSet.Next();                                  //记录下移
	}
	m_UserList.SetCurSel(0);                               //设置第一个用户为当前用户
	return TRUE;
}
```

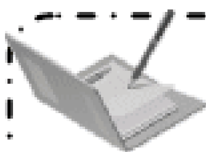
（4）在“确定”按钮的事件中实现用户名和密码的验证。代码如下：





Note

```
void CLoginDialog::OnLogin()
{
	CString sql,user,pass;
	m_UserList.GetWindowText(user);                         //获取用户名
	m_PassWord.GetWindowText(pass);                         //获取密码
❶	sql.Format("Select * From tab_User Where UserName = '%s' and PassWord = '%s'",
		user,pass);                                          //生成 SQL 查询语句
	m_DataSet.Open(sql);                                    //打开数据库
	if (m_DataSet.GetRecordCount() == 1)
	{
		::SetUserName(user);                                 //设置当前用户
❷		this->OnOK();
	}
	else
		AfxMessageBox("用户名或密码不正确！");
}
```



说明：

❶ Format 方法：该方法用于格式化字符串。

❷ OnOK 方法：该方法用于关闭当前窗口。

18.7　用户管理模块设计

18.7.1　用户管理模块概述

用户管理模块实现了对系统登录用户的添加、修改和删除操作。用户管理模块如图 18.21 所示。

18.7.2　用户管理技术分析

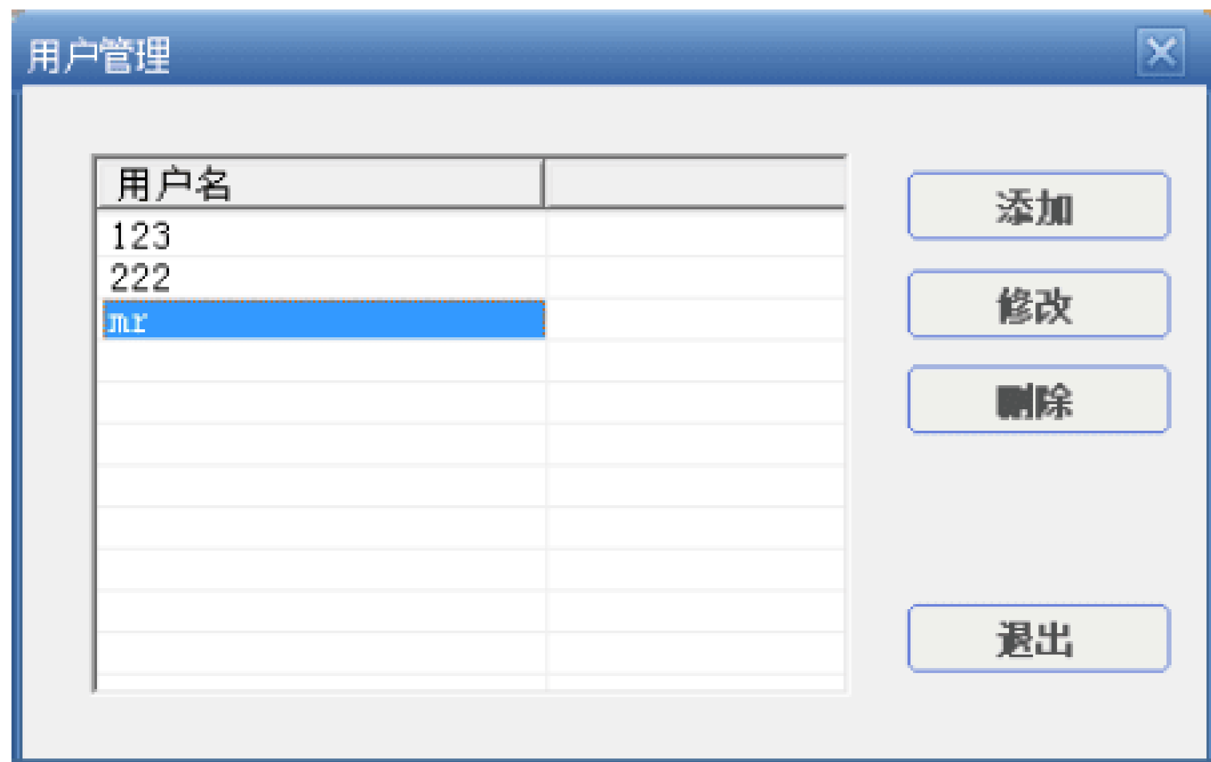
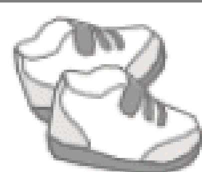


图 18.21　用户管理模块运行效果

在用户管理模块中使用 CListCtrl 控件显示用户信息，当对某一记录进行编辑或删除操作时必须要获取一个与记录对应的标识，所以在对用户列表进行添加时利用列表视图控件的 SetItemData 方法将记录集对应的行号添加到每一行对应的数据中。当对记录进行修改时就可以通过获取对应的行号对数据集中的数据进行修改了。获取数据时使用列表视图控件中的 GetItemData 方法。

（1）SetItemData 方法，该方法用于设置与指定项相关的 32 位应用指定的值。语法格式如下：



```
BOOL SetItemData(int nItem,DWORD dwData)
```

nItem 为要设定数据的列表项的索引。dwData 为与项相关联的 32 位值。

（2）GetItemData 方法，该方法用于获取与指定项相关的 32 位应用指定的值。语法格式如下：



```
DWORD GetItemData(int nItem) const
```

nItem 为要获取数据的列表项的索引值。

18.7.3 用户管理实现过程

本模块使用的数据表：Tab_User

（1）创建一个对话框，打开对话框属性窗口，将对话框的 ID 改为 IDD_DLGUSER，将对话框标题改为“用户管理”。

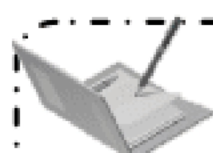
（2）向对话框中添加一个列表视图控件和 4 个按钮控件，各控件的属性设置如表 18.1 所示。

表 18.1 控件资源设置

控件 ID	控 件 属 性	对 应 变 量
IDC_LISTGRID	View：Report、Single selection	ClistCtrl m_grid
IDC_APPEND	Caption：添加	无
IDC_EDIT	Caption：修改	无
IDC_DELETE	Caption：删除	无
IDCANCEL	Caption：退出	无

（3）定义 UpdateGrid 方法，用来更新列表视图中显示的用户信息。实现代码如下：

```
void CUserManage::UpdateGrid()
{
    m_DataSet.Open("Select * From tab_User");                    //打开用户表
❶   m_grid.DeleteAllItems();                                      //清空列表视图中的全部记录
    for (int i = 0 ; i < m_DataSet.GetRecordCount();i++)          //循环记录集
    {
        //向列表视图中插入用户信息
❷       m_grid.InsertItem(i,(_bstr_t)m_DataSet.GetFields()->Item[L"UserName"]->Value);
        int no = m_DataSet.GetRecordNo();                         //获取当前记录集行号
        m_grid.SetItemData(i,no);                                 //存储列表视图中的项对应的行号
        m_DataSet.Next();                                         //行下移
    }
}
```

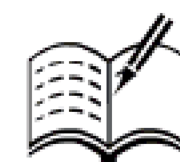


说明：

❶ DeleteAllItems 方法：该方法用于删除当前列表视图控件中所有的列表项。

❷ InsertItem 方法：该方法用于向列表视图控件中插入列表项。





（4）向对话框中添加 OnInitDialog 方法，在对话框的初始化方法中添加列表视图控件应显示的列头，并向列表视图控件中添加数据，代码如下：

```
BOOL CUserManage::OnInitDialog()
{
	CDialog::OnInitDialog();
	m_grid.SetExtendedStyle(LVS_EX_FULLROWSELECT|LVS_EX_GRIDLINES);	//列表控件样式
	m_grid.InsertColumn(0,"用户名");				//添加列
	m_grid.SetColumnWidth(0,150);				//设置列宽
	m_DataSet.SetConnection(::GetConnection());		//设置数据集的数据库连接对象
	UpdateGrid();						//向列表视图控件中添加数据
	return TRUE;
}
```

（5）在“添加”按钮的事件中弹出“用户编辑”窗体，输入用户名后单击“确定”按钮，实现对用户的添加，代码如下：

```
void CUserManage::OnAppend()
{
	CUserEdit useredit;					//定义用户编辑窗体
	if (useredit.DoModal() == IDOK)			//显示用户编辑窗体
	{
		m_DataSet.AddNew();				//数据集添加行
		//设置用户名字段的值为新用户
		m_DataSet.SetFieldValue("UserName",(_bstr_t)useredit.name);
		m_DataSet.Save();				//保存数据集
		UpdateGrid();					//更新列表视图控件中的数据
	}
}
```

（6）在“修改”按钮的事件中弹出“用户编辑”窗体，输入用户名后单击“确定”按钮，实现对用户的修改，代码如下：

```
void CUserManage::OnEdit()
{
	CUserEdit useredit;					//用户编辑窗体
	int no = m_grid.GetItemData(m_grid.GetSelectionMark());	//获取当前行记录行号
	m_DataSet.move(no-1);				//记录集指向指定行
	//获取用户名
	useredit.name = (char *)(_bstr_t)m_DataSet.GetFields()->Item[L"UserName"]->Value;
	if (useredit.DoModal() == IDOK)			//显示用户编辑窗体
	{
		m_DataSet.SetFieldValue("UserName",(_bstr_t)useredit.name);	//设置新的用户名
		m_DataSet.Save();				//保存数据
		UpdateGrid();					//更新列表
	}
}
```

（7）在“删除”按钮的单击事件中获取当前记录进行删除操作，代码如下：





Note

```
void CUserManage::OnDelete()
{
	if (MessageBox("是否删除此记录！","提示",
		MB_YESNO|MB_ICONWARNING) == IDYES)
	{
		int no = m_grid.GetItemData(m_grid.GetSelectionMark());	//获取记录集行号
		m_DataSet.move(no-1);					//移到指定行
		m_DataSet.Delete();					//删除
		m_DataSet.Save();					//保存
		UpdateGrid();						//更新列表
	}
}
```

18.7.4　单元测试

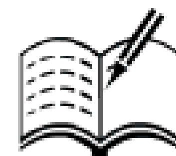
在测试用户管理模块时，曾出现这样的问题，由于用户在操作中不小心将所有的用户全部删除了，却没有重新创建新的用户，导致在下次登录时无法登录，下面来看一下原始的删除代码：

```
void CUserManage::OnDelete()
{
	if (MessageBox("是否删除此记录！","提示",
		MB_YESNO|MB_ICONWARNING) == IDYES)			//弹出消息提示
	{
		//删除用户所选中的用户信息
		int no = m_grid.GetItemData(m_grid.GetSelectionMark());
		m_DataSet.move(no-1);
		m_DataSet.Delete();
		m_DataSet.Save();
		UpdateGrid();
	}
}
```

为了解决上述问题，可以设置一个超级用户，该用户为“mr”，当用户要删除该用户时，提示不能删除，代码如下：

```
void CUserManage::OnDelete()
{
	int pos = m_grid.GetSelectionMark();			//获得当前选中项索引
	if (pos != -1)
	{
		CString name = m_grid.GetItemText(pos,0);		//获得当前选中项文本
		if (name != "mr")
		{
			if (MessageBox("是否删除此记录！","提示",
				MB_YESNO|MB_ICONWARNING) == IDYES)
			{
				int no = m_grid.GetItemData(m_grid.GetSelectionMark());
```






```
                m_DataSet.move(no-1);
                m_DataSet.Delete();
                m_DataSet.Save();
                UpdateGrid();
            }
        }
        else
        {
            MessageBox("该用户不能删除！");
            return;
        }
    }
}
```

18.8　部门管理模块设计

18.8.1　部门管理模块概述

部门管理记录了部门间的层次结构和部门信息，所以通常部门管理窗体中对于部门的显示是使用树列表显示的。部门管理模块如图 18.22 所示。

18.8.2　部门管理技术分析

由于部门通常都是存在层次级别的，所以在设计数据表结构时应至少创建 3 个字段：编号、父编号和名称。而在程序中显示部门信息时也是根据“父编号”作为查询条件不断地查找下一级的部门。

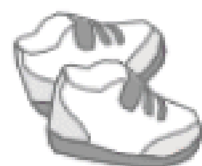


图 18.22　部门管理模块运行效果

在本系统中，由于部门信息通常不会太多，所以可以用嵌套的方式将部门信息一次性地读入树列表视图控件中。实现代码如下：

```
void CDeptManage::GetNode(HTREEITEM pNode,int nPid)
{
    HTREEITEM node;

    CADODataSet DataSet;                                         //定义记录集
    DataSet.SetConnection(::GetConnection());                    //设置数据库连接对象
    CString str;
    str.Format("Select * From tab_Dept where pid = %d",nPid);    //查询语句
    DataSet.Open(str);                                           //打开记录集
    int count = DataSet.GetRecordCount();                        //获取记录数量
```



```
        int ID;
        _variant_t value;
        for (int i = 0;i<count;i++)
        {
            node = m_tree.InsertItem((_bstr_t)DataSet.GetFields()->Item["DeptName"]->Value,
                    pNode);                                                    //部门名称
            value = (_variant_t)DataSet.GetFields()->Item["ID"]->Value;        //编号
            ID = value.intVal;
            m_tree.SetItemData(node,ID);                                       //与节点关联
            GetNode(node,ID);                                                  //获取子节点
            DataSet.Next();                                                    //记录下移
        }
}
```



Note

18.8.3 部门管理实现过程

本模块使用的数据表：tab_Dept

（1）创建一个对话框，打开对话框属性窗口，将对话框的 ID 改为 IDD_DLGDEPT，将对话框标题改为“部门管理”。

（2）向对话框中添加一个树列表控件、4 个按钮控件，各控件的属性设置如表 18.2 所示。

表 18.2 控件资源设置

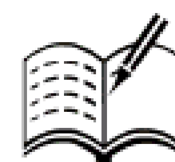
控件 ID	控 件 属 性	对 应 变 量
IDC_TREEDEPT	Has buttons、Has lines、Lines at root、Border	CTreeCtrl m_tree
IDC_APPEND	Caption：添加	无
IDC_EDIT	Caption：修改	无
IDC_DELETE	Caption：删除	无
IDCANCEL	Caption：退出	无

（3）定义 GetNode 方法用来按层级关系获取部门表中的所有数据，并添加到树列表控件中。该方法由 UpdateDept 方法进行调用，实现代码如下：

```
void CDeptManage::UpdateDept()
{
    m_tree.DeleteAllItems();                                  //清空树列表视图中的所有数据
    GetNode(TVI_ROOT,0);                                      //生成树列表
}

void CDeptManage::GetNode(HTREEITEM pNode,int nPid)
{
    HTREEITEM node;
    CADODataSet DataSet;                                      //定义记录集
    DataSet.SetConnection(::GetConnection());                 //设置数据库连接对象
    CString str;
    str.Format("Select * From tab_Dept where pid = %d",nPid); //查询语句
```





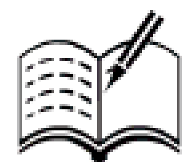

```
        DataSet.Open(str);                                                   //打开记录集
        int count = DataSet.GetRecordCount();                                //获取记录数量
        int ID;
        _variant_t value;
        for (int i = 0;i<count;i++)
        {
            node   =   m_tree.InsertItem((_bstr_t)DataSet.GetFields()->Item["DeptName"]->Value,pNode);
//部门名称
            value = (_variant_t)DataSet.GetFields()->Item["ID"]->Value;      //编号
            ID = value.intVal;
            m_tree.SetItemData(node,ID);                                     //与节点关联
            GetNode(node,ID);                                                //获取子节点
            DataSet.Next();                                                  //记录下移
        }
}
```

（4）当单击“添加”按钮时将弹出部门编辑窗体，输入部门信息后单击“确定”按钮将添加一个新的部门，代码如下：

```
void CDeptManage::OnAdd()
{
        CDeptEdit deptedit;                                                  //部门编辑
        if (deptedit.DoModal() == IDOK)                                      //显示部门编辑窗体
        {
            HTREEITEM pNode = m_tree.GetSelectedItem();                      //获取选中节点
            int pID;
            if (deptedit.isroot)                                             //根节点
                pID = 0;
            else
                pID = m_tree.GetItemData(pNode);                             //子节点
            CADODataSet dataset;                                             //定义记录集
            dataset.SetConnection(::GetConnection());                        //设置数据库连接对象
            dataset.Open("Select top 1 * From tab_Dept");                    //打开记录集
            dataset.AddNew();                                                //添加新记录
            dataset.SetFieldValue("DeptName",(_variant_t)deptedit.name);     //部门名称
            dataset.SetFieldValue("memo",(_variant_t)deptedit.memo);         //备注
            dataset.SetFieldValue("PID",(long)pID);                          //父编号
            dataset.Save();                                                  //保存
            UpdateDept();                                                    //更新树列表
        }
}
```

（5）当单击“修改”按钮时将弹出部门编辑窗体，输入部门信息后单击“确定”按钮将添加一个新的部门，代码如下：

```
void CDeptManage::OnEdit()
{
        CDeptEdit deptedit;                                                  //部门编辑窗体
        deptedit.visible = false;
        HTREEITEM pNode = m_tree.GetSelectedItem();                          //获取选中节点
        if (pNode == 0)
```


Note

```
        return;
    int pID = m_tree.GetItemData(pNode);                              //获取节点对应的编号
    CADODataSet dataset;                                              //定义记录集
    dataset.SetConnection(::GetConnection());                         //设置数据库连接
    CString str;
    str.Format("Select * From tab_Dept where id = %d",pID);           //生成查询语句
    dataset.Open(str);                                                //打开记录集
    deptedit.name = (char *)(_bstr_t)dataset.GetFields()->Item[L"DeptName"]->Value;   //部门名称
    deptedit.memo = (char *)(_bstr_t)dataset.GetFields()->Item["memo"]->Value;        //备注
    if (deptedit.DoModal() == IDOK)                                   //显示部门编辑窗体
    {
        dataset.SetFieldValue("DeptName",(_variant_t)deptedit.name); //部门名称
        dataset.SetFieldValue("memo",(_variant_t)deptedit.memo);     //备注
        dataset.Save();                                              //保存
        UpdateDept();                                                //更新树列表
    }
}
```

（6）当单击“删除”按钮时将删除当前选中的节点，代码如下：

```
void CDeptManage::OnDelete()
{
    HTREEITEM pNode = m_tree.GetSelectedItem();                       //获取选中节点
    if (pNode == 0)
        return;
    if (MessageBox("是否删除此记录！","提示",
        MB_YESNO|MB_ICONWARNING) == IDYES)
    {
        int pID = m_tree.GetItemData(pNode);                          //获取节点对应的编号
        CADODataSet dataset;                                          //定义记录集
        dataset.SetConnection(::GetConnection());                     //设置数据库连接
        CString str;
        str.Format("Select * From tab_Dept where id = %d",pID);       //生成查询语句
        dataset.Open(str);                                            //打开记录集
        dataset.Delete();                                             //删除记录
        dataset.Save();                                               //保存
        UpdateDept();                                                 //更新树列表
    }
}
```

18.9　人员信息管理模块设计

18.9.1　人员信息管理模块概述

人员信息管理模块根据部门分类显示，同时可对人员信息进行维护，人员信息管理模块如图 18.23 所示。





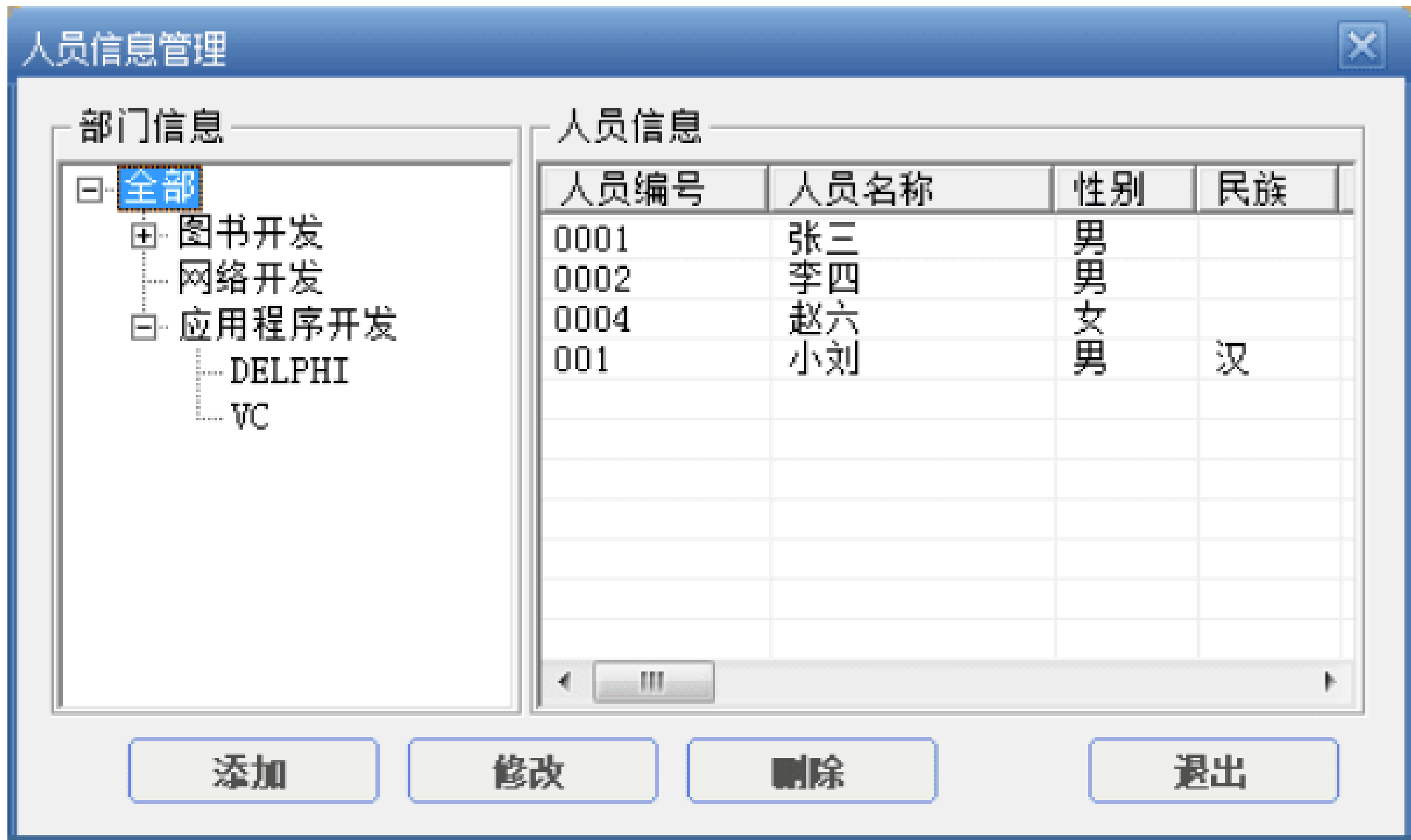


图 18.23　人员信息管理模块

18.9.2　人员信息管理技术分析

在人员信息管理界面中可以看到，窗体的左侧是部门信息，右侧是人员信息。当选中某一部门信息分类时右侧的人员信息会根据选中的部门进行人员信息的分类显示。这一操作主要是通过树列表视图控件中的 OnSelchanged 事件完成的，当树列表中的选中节点发生改变时就会触发该事件。实现代码如下：

```
void CPersonManage::OnSelchangedTreedept(NMHDR* pNMHDR, LRESULT* pResult)
{
	NM_TREEVIEW* pNMTreeView = (NM_TREEVIEW*)pNMHDR;            //获取树列表结构信息
	m_DeptID = m_tree.GetItemData(pNMTreeView->itemNew.hItem);   //获取部门编号
	UpdatePerson();                                             //更新人员信息
	*pResult = 0;
}
```

18.9.3　人员信息管理实现过程

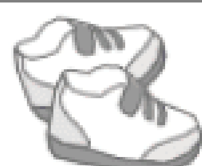
本模块使用的数据表：tab_Dept、tab_Employees

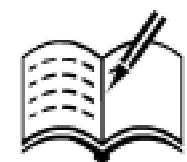
（1）创建一个对话框，打开对话框属性窗口，将对话框的 ID 改为 IDD_DLGPERSON，将对话框标题改为“人员信息管理”。

（2）向对话框中添加两个群组控件、一个树列表视图控件、一个列表视图控件和 4 个按钮控件，各控件的属性设置如表 18.3 所示。

表 18.3　控件资源设置

控件 ID	控 件 属 性	对 应 变 量
IDC_LISTPERSON	View ：Icon、Single selection	ClistCtrl　m_list
IDC_TREEDEPT	Has buttons、Has lines、Lines at root、Border	CTreeCtrl　m_tree
IDC_APPEND	Caption：添加	无





Note

续表

控件 ID	控 件 属 性	对 应 变 量
IDC_EDIT	Caption：修改	无
IDC_DELETE	Caption：删除	无
IDCANCEL	Caption：退出	无

（3）添加 GetNode 方法获取部门表中的数据信息添加到树列表视图控件中。该方法由 UpdateDept 方法调用。实现代码如下：

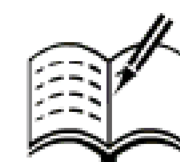
```
void CDeptManage::UpdateDept()
{
	m_tree.DeleteAllItems();                                    //清空树列表视图中的所有数据
	GetNode(TVI_ROOT,0);                                        //生成树列表
}

void CDeptManage::GetNode(HTREEITEM pNode,int nPid)
{
	HTREEITEM node;
	CADODataSet DataSet;                                        //定义记录集
	DataSet.SetConnection(::GetConnection());                   //设置数据库连接对象
	CString str;
	str.Format("Select * From tab_Dept where pid = %d",nPid);   //查询语句
	DataSet.Open(str);                                          //打开记录集
	int count = DataSet.GetRecordCount();                       //获取记录数量
	int ID;
	_variant_t value;
	for (int i = 0;i<count;i++)
	{
		node = m_tree.InsertItem((_bstr_t)DataSet.GetFields()->Item["DeptName"]->Value,pNode);
//部门名称
		value = (_variant_t)DataSet.GetFields()->Item["ID"]->Value;   //编号
		ID = value.intVal;
		m_tree.SetItemData(node,ID);                            //与节点关联
		GetNode(node,ID);                                       //获取子节点
		DataSet.Next();                                         //记录下移
	}
}
```

（4）定义 UpdatePerson 方法用来更新人员信息，将其显示在列表视图控件中，代码如下：

```
void CPersonManage::UpdatePerson()
{
	m_list.DeleteAllItems();                                    //清空列表视图控件中的数据
	CADODataSet DataSet;                                        //定义记录集
	DataSet.SetConnection(::GetConnection());                   //设置数据库连接对象
	CString str;
	if (m_DeptID == -1)
		str.Format("Select * From tab_Employees");              //显示所有人员信息
```



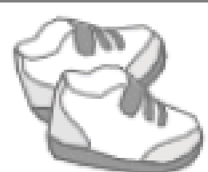



```
        else                                                    //显示指定部门人员信息
            str.Format("Select * From tab_Employees where Dept = %d",m_DeptID);
        DataSet.Open(str);                                      //打开记录集
        int count = DataSet.GetRecordCount();                   //获取记录集记录数量
        int n = 0;
        _variant_t value;
        for (int i = 0;i<count;i++)                             //循环记录集
        {
            int index = 1;
            //人员编号
            m_list.InsertItem(n,(_bstr_t)DataSet.GetFields()->Item["Emp_Id"]->Value);
            value = DataSet.GetFields()->Item["AutoID"]->Value;     //自动编号
            m_list.SetItemData(n,value.lVal);                   //将自动编号与列表中的项关联
            //名称
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Emp_NAME"]->Value);
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Sex"]->Value);//性别
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Nationality"]->Value);
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Birth"]->Value);
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Political_Party"]->Value);
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Culture_Level"]->Value);

m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Marital_Condition"]->Value);
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Id_Card"]->Value);
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Office_phone"]->Value);
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Mobile"]->Value);
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["HireDate"]->Value);
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Duty"]->Value);
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Memo"]->Value);
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Files_Keep_Org"]->Value);
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Hukou"]->Value);
m_list.SetItemText(n,index++,(_bstr_t)DataSet.GetFields()->Item["Family_Place"]->Value);
n++;
DataSet.Next();                                                 //记录下移
        }
}
```

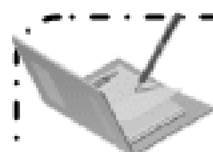
（5）添加 OnInitDialog 方法，用于初始化人员信息管理窗体中的数据。在该方法中显示部门信息、人员信息，代码如下：

```
BOOL CPersonManage::OnInitDialog()
{
        CDialog::OnInitDialog();
        m_DeptID = -1;
        UpdateDept();
        int i = 0;
❶       m_list.InsertColumn(i,"人员编号");
❷       m_list.SetColumnWidth(i++,80);
        m_list.InsertColumn(i,"人员名称");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"性别");
```






```
        m_list.SetColumnWidth(i++,50);
        m_list.InsertColumn(i,"民族");
        m_list.SetColumnWidth(i++,50);
        m_list.InsertColumn(i,"出生日期");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"政治面貌");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"文化程度");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"婚姻状况");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"身份证号");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"办公电话");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"手机电话");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"到岗日期");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"职务");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"备注");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"家庭住址");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"档案所在地");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"户口所在地");
        m_list.SetColumnWidth(i++,100);
        m_list.SetExtendedStyle(LVS_EX_FULLROWSELECT|LVS_EX_GRIDLINES);
        UpdatePerson();
        return TRUE;
}
```



说明：

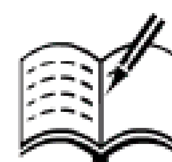
❶ InsertColumn 方法：该方法用于向当前列表视图控件中插入列标题。

❷ SetColumnWidth 方法：该方法用于设置列表视图控件的扩展风格。

（6）单击“添加”按钮，弹出人员编辑窗体，输入人员信息后单击“保存”按钮实现人员信息添加，代码如下：

```
void CPersonManage::OnAdd()
{
        CPersonEdit personedit;
        personedit.m_DeptData = m_DeptID;
        if (personedit.DoModal() == IDOK)
        {
```






```
            CADODataSet dataset;
            dataset.SetConnection(::GetConnection());
            CString str = "select top 1 * from tab_Employees";
            dataset.Open(str);
            dataset.AddNew();
            dataset.SetFieldValue("Emp_Id",(_bstr_t)personedit.m_id);
            dataset.SetFieldValue("Emp_NAME",(_bstr_t)personedit.m_name);
            dataset.SetFieldValue("Sex",(_bstr_t)personedit.m_sex);
            dataset.SetFieldValue("Nationality",(_bstr_t)personedit.m_nationality);
            dataset.SetFieldValue("Birth",(_bstr_t)personedit.m_birth.Format("%Y-%m-%d"));
            dataset.SetFieldValue("Political_Party",(_bstr_t)personedit.m_farty);
            dataset.SetFieldValue("Culture_Level",(_bstr_t)personedit.m_culture);
            dataset.SetFieldValue("Marital_Condition",(_bstr_t)personedit.m_marital);
            dataset.SetFieldValue("Id_Card",(_bstr_t)personedit.m_card);
            dataset.SetFieldValue("Office_phone",(_bstr_t)personedit.m_office);
            dataset.SetFieldValue("Mobile",(_bstr_t)personedit.m_mobile);
            dataset.SetFieldValue("HireDate",(_bstr_t)personedit.m_hire.Format("%Y-%m-%d"));
            dataset.SetFieldValue("Duty",(_bstr_t)personedit.m_duty);
            dataset.SetFieldValue("Memo",(_bstr_t)personedit.m_memo);
            dataset.SetFieldValue("Files_Keep_Org",(_bstr_t)personedit.m_files);
            dataset.SetFieldValue("Hukou",(_bstr_t)personedit.m_hukou);
            dataset.SetFieldValue("Family_Place",(_bstr_t)personedit.m_family);
            dataset.SetFieldValue("dept",personedit.m_DeptData);
            dataset.Save();
            UpdatePerson();
      }
}
```

（7）单击“修改”按钮，弹出人员编辑窗体，输入人员信息后单击“保存”按钮实现人员信息的修改，代码如下：

```
void CPersonManage::OnEdit()
{
      if (m_list.GetSelectionMark() == -1)
            return;
      int id = m_list.GetItemData(m_list.GetSelectionMark());
      CPersonEdit personedit;
      CADODataSet dataset;
      dataset.SetConnection(::GetConnection());
      CString str;
      str.Format("select * from tab_Employees where autoid = %d",id);
      dataset.Open(str);
      personedit.m_id = (char *)(_bstr_t)dataset.GetFields()->Item["Emp_Id"]->Value;
      personedit.m_name = (char *)(_bstr_t)dataset.GetFields()->Item["Emp_NAME"]->Value;
      personedit.m_sex = (char *)(_bstr_t)dataset.GetFields()->Item["Sex"]->Value;
      personedit.m_nationality = (char *)(_bstr_t)dataset.GetFields()->Item["Nationality"]->Value;
      CString birth = (char *)(_bstr_t)dataset.GetFields()->Item["Birth"]->Value;
      if (!birth.IsEmpty())
      {
            //设置日期数据
```





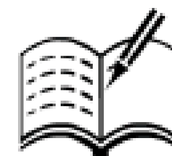
Note

```
        int yy=atoi(birth.Left(4));
        int mm=atoi(birth.Mid(6,2));
        int dd=atoi(birth.Mid(9,2));
        CTime tbirth(yy,mm,dd,0,0,0);
        personedit.m_birth = tbirth;
}
personedit.m_farty = (char *)(_bstr_t)dataset.GetFields()->Item["Political_Party"]->Value;
personedit.m_culture = (char *)(_bstr_t)dataset.GetFields()->Item["Culture_Level"]->Value;
personedit.m_marital = (char *)(_bstr_t)dataset.GetFields()->Item["Marital_Condition"]->Value;
personedit.m_card = (char *)(_bstr_t)dataset.GetFields()->Item["Id_Card"]->Value;
personedit.m_office = (char *)(_bstr_t)dataset.GetFields()->Item["Office_phone"]->Value;
personedit.m_mobile = (char *)(_bstr_t)dataset.GetFields()->Item["Mobile"]->Value;
CString hire = (char *)(_bstr_t)dataset.GetFields()->Item["HireDate"]->Value;
if (!hire.IsEmpty())
{
        //设置日期数据
        int yy=atoi(hire.Left(4));
        int mm=atoi(hire.Mid(6,2));
        int dd=atoi(hire.Mid(9,2));
        CTime thire(yy,mm,dd,0,0,0);
        personedit.m_hire = thire;
}
personedit.m_duty = (char *)(_bstr_t)dataset.GetFields()->Item["Duty"]->Value;
personedit.m_memo = (char *)(_bstr_t)dataset.GetFields()->Item["Memo"]->Value;
personedit.m_files = (char *)(_bstr_t)dataset.GetFields()->Item["Files_Keep_Org"]->Value;
personedit.m_hukou = (char *)(_bstr_t)dataset.GetFields()->Item["Hukou"]->Value;
personedit.m_family = (char *)(_bstr_t)dataset.GetFields()->Item["Family_Place"]->Value;
personedit.m_DeptData = dataset.GetFields()->Item["Dept"]->Value;
if (personedit.DoModal() == IDOK)
{
        dataset.SetFieldValue("Emp_Id",(_bstr_t)personedit.m_id);
        dataset.SetFieldValue("Emp_NAME",(_bstr_t)personedit.m_name);
        dataset.SetFieldValue("Sex",(_bstr_t)personedit.m_sex);
        dataset.SetFieldValue("Nationality",(_bstr_t)personedit.m_nationality);
        dataset.SetFieldValue("Birth",(_bstr_t)personedit.m_birth.Format("%Y-%m-%d"));
        dataset.SetFieldValue("Political_Party",(_bstr_t)personedit.m_farty);
        dataset.SetFieldValue("Culture_Level",(_bstr_t)personedit.m_culture);
        dataset.SetFieldValue("Marital_Condition",(_bstr_t)personedit.m_marital);
        dataset.SetFieldValue("Id_Card",(_bstr_t)personedit.m_card);
        dataset.SetFieldValue("Office_phone",(_bstr_t)personedit.m_office);
        dataset.SetFieldValue("Mobile",(_bstr_t)personedit.m_mobile);
        dataset.SetFieldValue("HireDate",(_bstr_t)personedit.m_hire.Format("%Y-%m-%d"));
        dataset.SetFieldValue("Duty",(_bstr_t)personedit.m_duty);
        dataset.SetFieldValue("Memo",(_bstr_t)personedit.m_memo);
        dataset.SetFieldValue("Files_Keep_Org",(_bstr_t)personedit.m_files);
        dataset.SetFieldValue("Hukou",(_bstr_t)personedit.m_hukou);
        dataset.SetFieldValue("Family_Place",(_bstr_t)personedit.m_family);
        dataset.SetFieldValue("dept",personedit.m_DeptData);
        dataset.Save();
```



```
            UpdatePerson();
        }
}
```

（8）单击“删除”按钮实现当前选中的人员信息记录删除的操作，代码如下：




```
void CPersonManage::OnDelete()
{
    if (MessageBox("是否删除此记录！ ","提示",MB_YESNO|MB_ICONWARNING) == IDYES)
    {
        if (m_list.GetSelectionMark() == -1)
            return;
        int id = m_list.GetItemData(m_list.GetSelectionMark());
        CADODataSet dataset;
        dataset.SetConnection(::GetConnection());
        CString str;
        str.Format("select * from tab_Employees where autoid = %d",id);
        dataset.Open(str);                                          //打开表
        dataset.Delete();                                           //删除记录
        dataset.Save();                                             //保存操作
        UpdatePerson();
    }
}
```

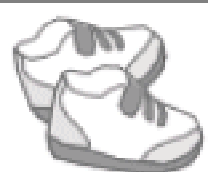
18.10　考勤管理模块设计

18.10.1　考勤管理模块概述

考勤管理模块将所有人员当天的考勤信息录入到该模块中，并且可以根据年、月和人员对已录入的考勤记录进行查询。人事考勤管理模块如图 18.24 所示。



图 18.24　考勤管理模块



18.10.2 考勤管理技术分析



Note

在进行程序设计时日期型数据可以使用字符串的形式存入日期类型的数据库字段中，相反地，字符串类型的日期数据要想转换成日期类型的数据就必须自己实现其转换功能。在该模块中实现了字符串形式的日期和时间分别转换成日期类型的数据。

GetTimeForStr 方法用来将字符串形式的时间转换成日期类型。实现代码如下：

```
CTime CCheckManage::GetTimeForStr(CString timestr)
{
	int h,m,s;
	if (timestr.GetLength() < 8)                        //不足 8 位补 0
		timestr = "0"+timestr;
	h = atoi(timestr.Left(2));                          //获取小时
	m = atoi(timestr.Mid(3,2));                         //获取分
	s = atoi(timestr.Right(2));                         //获取秒
	CTime result(2000,1,1,h,m,s);                       //生成日期
	return result;
}
```

GetDateForStr 方法用来将字符串类型的日期值转换成日期类型的数据。实现代码如下：

```
CTime CCheckManage::GetDateForStr(CString datestr)
{
	int y,m,d;
	y = atoi(datestr.Left(4));                          //年
	m = atoi(datestr.Mid(5,2));                         //月
	d = abs(atoi(datestr.Right(2)));                    //日
	CTime result(y,m,d,8,0,0);                          //生成日期
	return result;
}
```

在该模块中还实现了一个时间相减的方法，在这个方法中实现相减都是转换成秒后进行减法计算的，然后再将秒转换成对应时间类型数据。实现代码如下：

```
CTime CCheckManage::DecTime(CTime one, CTime two)
{
	int yy,mm,dd,h,s,m,onetemp,twotemp;
	yy = 2000;//one.GetYear();// - two.GetYear();
	mm = 1;
	dd = 1;
	onetemp = one.GetSecond() + one.GetMinute() * 60 + one.GetHour() * 60 * 60;        //总秒数
	twotemp = two.GetSecond() + two.GetMinute() * 60 + two.GetHour() * 60 * 60;
	if ((onetemp - twotemp) < 0)
	{
		h = m = s = 0;
	}
	else
```



```
    {
        h = (onetemp - twotemp) / 60 / 60;                          //小时
        m = ((onetemp - twotemp) - h * 60 * 60) / 60;               //分种
        s = ((onetemp - twotemp) - h * 60 * 60) - m * 60;           //秒
    }
    CTime time (yy,mm,dd,h,m,s);                                    //生成时间数据
    return time;
}
```



Note

18.10.3　考勤管理实现过程

本模块使用的数据表：tab_Check

（1）创建一个对话框，打开对话框属性窗口，将对话框的 ID 改为 IDD_DLGCHECK，将对话框标题改为“考勤管理”。

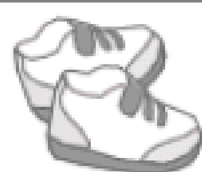
（2）向对话框中添加一个复选框控件、3 个静态文本框控件、3 个组合框控件、4 个按钮控件和一个列表视图控件，各控件的属性设置如表 18.4 所示。

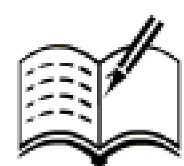
表 18.4　控件资源设置

控件 ID	控 件 属 性	对 应 变 量
IDC_CHECK1	Caption：全部显示	BOOL　m_check
IDC_COMBOYY	Type：Drop List	CComboBox　m_cyy CString　m_yy
IDC_COMBOMM	Type：Drop List	CComboBox　m_cmm CString　m_mm
IDC_COMBOEMP	Type：Drop List	CComboBox　m_cemp CString　m_emp
IDC_LISTPERSON	View　：Icon、Single selection	ClistCtrl　m_list
IDC_APPEND	Caption：添加	无
IDC_EDIT	Caption：修改	无
IDC_DELETE	Caption：删除	无
IDCANCEL	Caption：退出	无

（3）添加 UpdateList 方法，用于更新显示人员的考勤信息。实现代码如下：

```
void CCheckManage::UpdateList()
{
    this->UpdateData();
    CString str;
    if (m_check)
        str.Format("Select * From tab_Check");                  //显示所有员工考勤信息
    else
    {
        CString Starttime,EndTime;
```





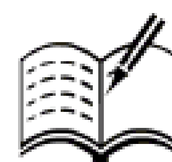
Note

```
            Starttime = m_yy + "-" + m_mm + "-1" ;
            EndTime.Format("DATEADD(month,1,'%s')",Starttime);
            if (m_emp == "(全部)")                                    //显示指定时间内的所有员工考勤信息
                str.Format("Select * From tab_Check where checkdate between '%s' and %s",Starttime,
EndTime);
            else                                                       //显示指定时间内的某个员工考勤信息
                str.Format("Select * From tab_Check where name = '%s' and \
                    checkdate between '%s' and %s",m_emp,Starttime,EndTime);
        }
        CADODataSet dataset;                                           //定义记录集
        dataset.SetConnection(::GetConnection());                      //设置数据库连接对象
        dataset.Open(str);                                             //打开记录集
        m_list.DeleteAllItems();                                       //清空列表视图控件中的所有数据
        for (int i = 0 ; i < dataset.GetRecordCount() ; i++)
        {
            int n = 0;
            long data = dataset.GetFields()->Item["autoid"]->Value;
            m_list.InsertItem(i,"");
            m_list.SetItemData(i,data);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["name"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["ondutytime"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["offdutytime"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["ontime"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["offtime"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["leave"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["onleave"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["offleave"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["latetime"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["leaveearly"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["memo"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["checkdate"]->Value);
            dataset.Next();                                            //记录下移
        }
}
```

（4）向对话框中添加 OnInitDialog 方法，在对话框初始化时设置列表视图控件的表头和列宽度，以及查询条件选择控件的设置，实现代码如下：

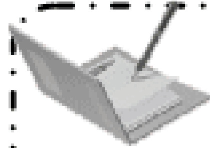
```
BOOL CCheckManage::OnInitDialog()
{
        CDialog::OnInitDialog();
        int i = 0;
        m_list.InsertColumn(i,"人员姓名");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"上班时间");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"下班时间");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"上班考勤时间");
        m_list.SetColumnWidth(i++,100);
```






```
    m_list.InsertColumn(i,"下班考勤时间");
    m_list.SetColumnWidth(i++,100);
    m_list.InsertColumn(i,"请假类别");
    m_list.SetColumnWidth(i++,100);
    m_list.InsertColumn(i,"请假起始时间");
    m_list.SetColumnWidth(i++,100);
    m_list.InsertColumn(i,"请假结束时间");
    m_list.SetColumnWidth(i++,100);
    m_list.InsertColumn(i,"迟到时间");
    m_list.SetColumnWidth(i++,100);
    m_list.InsertColumn(i,"早退时间");
    m_list.SetColumnWidth(i++,100);
    m_list.InsertColumn(i,"备注");
    m_list.SetColumnWidth(i++,100);
    m_list.InsertColumn(i,"考勤日期");
    m_list.SetColumnWidth(i++,130);
    m_list.SetExtendedStyle(LVS_EX_FULLROWSELECT|LVS_EX_GRIDLINES);
    m_check = true;
    this->UpdateData(false);
    int curyear,curmonth;                               //当前年、月
❶   CTime time(CTime::GetCurrentTime());
❷   curyear = time.GetYear();                           //当前年
    curmonth = time.GetMonth();                         //当前月
    char value[10];
    for (int y = 2000; y < 2100 ;y++)                   //向年列表中添加数值
    {
        _itoa(y,value,10);
        m_cyy.InsertString(y-2000,value);
    }
    m_cyy.SetCurSel(curyear-2000);
    for (int n = 1 ; n<=12 ;n++)                        //向月列表中添加数值
    {
        _itoa(n,value,10);
        m_cmm.InsertString(n-1,value);
    }
    m_cmm.SetCurSel(curmonth-1);
    CADODataSet dataset;
    dataset.SetConnection(::GetConnection());
    dataset.Open("Select * From tab_Employees");        //打开员工信息表
    m_cemp.InsertString(0,"(全部)");
    //向员工列表中添加员工信息
    for (int index = 1 ; index < dataset.GetRecordCount() ; index++)
    {       m_cemp.InsertString(index,(_bstr_t)dataset.GetFields()->Item["emp_name"]->Value);
        dataset.Next();
    }
    m_cemp.SetCurSel(0);
    UpdateList();                                       //更新考勤信息列表
    return TRUE;
}
```





说明：

❶ GetCurrentTime 方法：该方法用于获得当前系统时间。

❷ GetYear 方法：该方法用于获得 CTime 对象中的年数据。



Note

（5）添加 OnAdd 方法，用于向考勤信息表中添加员工的日考勤数据，实现代码如下：

```
void CCheckManage::OnAdd()
{
	CCheckEdit checkedit;                                    //员工考勤编辑窗体
	if (checkedit.DoModal() == IDOK)                         //显示员工考勤编辑窗体
	{
		CString time;
		CString str = "Select top 1 * From tab_Check";
		CADODataSet dataset;
		dataset.SetConnection(::GetConnection());
		dataset.Open(str);                                   //打开员工考勤表
		dataset.AddNew();                                    //添加新记录
		dataset.SetFieldValue("name",(_bstr_t)checkedit.m_name);
		dataset.SetFieldValue("checkdate",(_bstr_t)checkedit.m_datecheck.Format("%Y-%m-%d"));
		dataset.SetFieldValue("ondutytime",(_bstr_t)checkedit.m_timeonduty.Format("%H:%M:%S"));
		dataset.SetFieldValue("offdutytime",(_bstr_t)checkedit.m_timeoffduty.Format("%H:%M:%S"));
		dataset.SetFieldValue("ontime",(_bstr_t)checkedit.m_timeon.Format("%H:%M:%S"));
		dataset.SetFieldValue("offtime",(_bstr_t)checkedit.m_timeoff.Format("%H:%M:%S"));
		dataset.SetFieldValue("leave",(_bstr_t)checkedit.m_leave);
		dataset.SetFieldValue("onleave",(_bstr_t)checkedit.m_timeonleave.Format("%H:%M:%S"));
		dataset.SetFieldValue("offleave",(_bstr_t)checkedit.m_timeoffleave.Format("%H:%M:%S"));
		dataset.SetFieldValue("memo",(_bstr_t)checkedit.m_memo);
		CTime latetime = DecTime(checkedit.m_timeon,checkedit.m_timeonduty);//时间相减
		time.Format("%d:%d:%d",latetime.GetHour(),latetime.GetMinute(),latetime.GetSecond());
		dataset.SetFieldValue("latetime",(_bstr_t)time);
		CTime leaveearly = DecTime(checkedit.m_timeoff,checkedit.m_timeoffduty);
		time.Format("%d:%d:%d",leaveearly.GetHour(),leaveearly.GetMinute(),leaveearly.GetSecond());
		dataset.SetFieldValue("leaveearly",(_bstr_t)time);
		dataset.Save();                                      //保存记录
		UpdateList();                                        //更新考勤信息列表
	}
}
```

（6）添加 OnEdit 方法，用于编辑考勤信息表中员工的日考勤数据，实现代码如下：

```
void CCheckManage::OnEdit()
{
	if (m_list.GetSelectionMark() == -1)                     //判断是否存在选中记录
		return;
	int id = m_list.GetItemData(m_list.GetSelectionMark());  //获取记录唯一标识
	CCheckEdit checkedit;                                    //考勤信息编辑窗体
	CString str;
	str.Format("Select * From tab_Check where autoid = %d",id);
	CADODataSet dataset;
```






```
        dataset.SetConnection(::GetConnection());
        dataset.Open(str);                                          //打开记录集
        checkedit.m_name = (char *)(_bstr_t)dataset.GetFields()->Item["name"]->Value;
        checkedit.m_timeonduty = GetTimeForStr((char*)(_bstr_t)dataset.GetFields()->Item["ondutytime"]-> Value);
        checkedit.m_timeoffduty = GetTimeForStr((char*)(_bstr_t)dataset.GetFields()->Item["offdutytime"]-> Value);
        checkedit.m_timeon = GetTimeForStr((char*)(_bstr_t)dataset.GetFields()->Item["ontime"]->Value);
        checkedit.m_timeoff = GetTimeForStr((char*)(_bstr_t)dataset.GetFields()->Item["offtime"]->Value);
        checkedit.m_leave = (char *)(_bstr_t)dataset.GetFields()->Item["leave"]->Value;
        checkedit.m_timeonleave = GetTimeForStr((char*)(_bstr_t)dataset.GetFields()->Item["onleave"]->Value);
        checkedit.m_timeoffleave = GetTimeForStr((char*)(_bstr_t)dataset.GetFields()->Item["offleave"]->Value);
        checkedit.m_memo = (char *)(_bstr_t)dataset.GetFields()->Item["memo"]->Value;
        checkedit.m_datecheck = GetDateForStr((char*)(_bstr_t)dataset.GetFields()->Item["checkdate"]->Value);
        if (checkedit.DoModal() == IDOK)                            //显示考勤信息编辑窗体
        {
                CString time;
                dataset.SetFieldValue("name",(_bstr_t)checkedit.m_name);
                dataset.SetFieldValue("checkdate",(_bstr_t)checkedit.m_datecheck.Format("%Y-%m-%d"));
                dataset.SetFieldValue("ondutytime",(_bstr_t)checkedit.m_timeonduty.Format("%H:%M:%S"));
                dataset.SetFieldValue("offdutytime",(_bstr_t)checkedit.m_timeoffduty.Format("%H:%M:%S"));
                dataset.SetFieldValue("ontime",(_bstr_t)checkedit.m_timeon.Format("%H:%M:%S"));
                dataset.SetFieldValue("offtime",(_bstr_t)checkedit.m_timeoff.Format("%H:%M:%S"));
                dataset.SetFieldValue("leave",(_bstr_t)checkedit.m_leave);
                dataset.SetFieldValue("onleave",(_bstr_t)checkedit.m_timeonleave.Format("%H:%M:%S"));
                dataset.SetFieldValue("offleave",(_bstr_t)checkedit.m_timeoffleave.Format("%H:%M:%S"));
                dataset.SetFieldValue("memo",(_bstr_t)checkedit.m_memo);
                CTime latetime = DecTime(checkedit.m_timeon,checkedit.m_timeonduty);
                time.Format("%d:%d:%d",latetime.GetHour(),latetime.GetMinute(),latetime.GetSecond());
                dataset.SetFieldValue("latetime",(_bstr_t)time);
                CTime leaveearly = DecTime(checkedit.m_timeoffduty,checkedit.m_timeoff);
                time.Format("%d:%d:%d",leaveearly.GetHour(),leaveearly.GetMinute(),leaveearly.GetSecond());
                dataset.SetFieldValue("leaveearly",(_bstr_t)time);
                dataset.Save();                                     //保存记录集修改
                UpdateList();                                       //更新考勤信息列表
        }
}
```

（7）添加 OnDelete 方法，用于删除当前选择的考勤记录，实现代码如下：

```
void CCheckManage::OnDelete()
{
        if (MessageBox("是否删除此记录！","提示",
                MB_YESNO|MB_ICONWARNING) == IDYES)
        {
                if (m_list.GetSelectionMark() == -1)
                        return;
                int id = m_list.GetItemData(m_list.GetSelectionMark());
                CADODataSet dataset;
                dataset.SetConnection(::GetConnection());
                CString str;
                str.Format("select * from tab_Check where autoid = %d",id);
```

```
            dataset.Open(str);
            dataset.Delete();
            dataset.Save();
            UpdateList();
        }
}
```



Note

18.11 考勤汇总查询模块设计

18.11.1 考勤汇总查询模块概述

考勤汇总查询模块用于将日常录入的员工考勤信息根据时间范围和人员进行汇总查询，并显示员工的月出勤天数、迟到天数、请假天数等。考勤汇总查询模块如图 18.25 所示。

考勤汇总查询

年：2009 月：5 员工：张三 退出

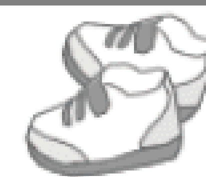
人员姓名	工作总天数	迟到总天数	早退总天数	病假总天数	事假总
张三	.83	.27	.02	0	0

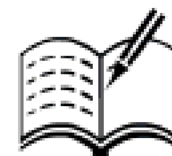
图 18.25 考勤汇总查询模块

18.11.2 考勤汇总查询技术分析

在该模块中汇总查询是通过 SQL 语句来实现的，这个汇总查询主要通过一些 SQL 子句组成一个 SQL 汇总查询语句。这些子句分别用来获取员工工作总天数、迟到总天数、早退总天数、病假总天数和事假总天数。

在进行天数计算时是将考勤时间转换成秒，先计算所使用的总秒数，最后再根据总秒数计算出天数。考勤汇总查询的 SQL 语句如下：






```
CString str,temp,where,datestr,StartDate,EndDate;
StartDate = m_yy + "-" + m_mm + "-1" ;
EndDate.Format("DATEADD(month,1,'%s')",StartDate);
datestr.Format(" between '%s' and %s",StartDate,EndDate);
temp += "select emp.emp_name ,ROUND(isnull(works.workday,0),2)";
temp += " workday,ROUND(isnull(lates.lateday,0),2) lateday,";
temp += " ROUND(isnull(leaveearlys.leaveearlyday,0),2) leaveearlyday,";
temp += " ROUND(isnull(bjdays.bjday,0),2) bjday,ROUND(isnull(sjdays.sjday,0),2) sjday";
temp += " from tab_Employees emp ";
temp += " left join";
temp += " (select sum(DATEDIFF(second,ontime,offtime)) / 60.0 / 60.0 / 8.0";
temp += " as workday,name    From tab_Check where checkdate %s group by name)";
temp += " works on emp.emp_name = works.name";
temp += " left join";
temp += " (select (sum(DATEPART(Hour,latetime)) * 60 * 60 + ";
temp += " sum(DATEPART(minute,latetime)) * 60 + sum(DATEPART(second,latetime)))";
temp += " /60.0 /60.0 /8.0 as lateday,name     From tab_Check where checkdate";
temp += " %s group by name) lates on emp.emp_name = lates.name";
temp += " left join";
temp += " (select (sum(DATEPART(Hour,leaveearly)) * 60 * 60 + ";
temp += " sum(DATEPART(minute,leaveearly)) * 60 + sum(DATEPART(second,leaveearly)))";
temp += " /60.0 /60.0 /8.0   as leaveearlyday,name     From tab_Check where ";
temp += " checkdate %s group by name) leaveearlys on emp.emp_name";
temp += " = leaveearlys.name";
temp += " left join";
temp += " (select isnull(sum(DATEDIFF(second,onleave,offleave))";
temp += " / 60.0 / 60.0 / 8.0,0) as bjday,name    From tab_Check where";
temp += " leave = '病假' and checkdate %s group by name) ";
temp += " bjdays on emp.emp_name = bjdays.name";
temp += " left join";
temp += " (select isnull(sum(DATEDIFF(second,onleave,offleave)) ";
temp += " / 60.0 / 60.0 / 8.0,0) as sjday,name    From tab_Check where ";
temp += " leave = '事假' and checkdate %s group by name) ";
temp += " sjdays on emp.emp_name = sjdays.name";
temp += " %s";
```

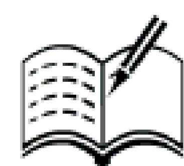
18.11.3　考勤汇总查询实现过程

本模块使用的数据表：tab_Employees、tab_Check

（1）创建一个对话框，打开对话框属性窗口，将对话框的 ID 改为 IDD_DLGCHECKSUM，将对话框标题改为“考勤汇总查询”。

（2）向对话框中添加 3 个静态文本框控件、3 个组合框控件、一个按钮控件和一个列表视图控件，各控件的属性设置如表 18.5 所示。





Note

表 18.5　控件资源设置

控件 ID	控 件 属 性	对 应 变 量
IDC_CYY	Type：Drop List	CComboBox　m_cyy CString　m_yy
IDC_CMM	Type：Drop List	CComboBox　m_cmm CString　m_mm
IDC_CEMP	Type：Drop List	CComboBox　m_cemp CString　m_emp
IDCANCEL	IDCANCEL	Caption：退出
IDC_LISTEMP	View ：Icon、Single selection	CListCtrl　m_list

（3）添加 UpdateList 方法，用于更新考勤汇总查询的数据。实现代码如下：

```
void CCheckSum::UpdateList()
{
	m_list.DeleteAllItems();
	this->UpdateData();
	CADODataSet dataset;
	dataset.SetConnection(::GetConnection());
	CString str,temp,where,datestr,StartDate,EndDate;
	StartDate = m_yy + "-" + m_mm + "-1" ;
	EndDate.Format("DATEADD(month,1,'%s')",StartDate);
	datestr.Format(" between '%s' and %s",StartDate,EndDate);
	temp += "select emp.emp_name ,ROUND(isnull(works.workday,0),2)";
	temp += " workday,ROUND(isnull(lates.lateday,0),2) lateday,";
	temp += " ROUND(isnull(leaveearlys.leaveearlyday,0),2) leaveearlyday,";
	temp += " ROUND(isnull(bjdays.bjday,0),2) bjday,ROUND(isnull(sjdays.sjday,0),2) sjday";
	temp += " from tab_Employees emp ";
	temp += " left join";
	temp += " (select sum(DATEDIFF(second,ontime,offtime)) / 60.0 / 60.0 / 8.0";
	temp += " as workday,name   From tab_Check where checkdate %s group by name)";
	temp += " works on emp.emp_name = works.name";
	temp += " left join";
	temp += " (select (sum(DATEPART(Hour,latetime)) * 60 * 60 + ";
	temp += " sum(DATEPART(minute,latetime)) * 60 + sum(DATEPART(second,latetime)))";
	temp += " /60.0 /60.0 /8.0 as lateday,name    From tab_Check where checkdate";
	temp += " %s group by name) lates on emp.emp_name = lates.name";
	temp += " left join";
	temp += " (select (sum(DATEPART(Hour,leaveearly)) * 60 * 60 + ";
	temp += " sum(DATEPART(minute,leaveearly)) * 60 + sum(DATEPART(second,leaveearly)))";
	temp += " /60.0 /60.0 /8.0   as leaveearlyday,name    From tab_Check where ";
	temp += " checkdate %s group by name) leaveearlys on emp.emp_name";
	temp += " = leaveearlys.name";
	temp += " left join";
	temp += " (select isnull(sum(DATEDIFF(second,onleave,offleave))";
	temp += " / 60.0 / 60.0 / 8.0,0) as bjday,name   From tab_Check where";
	temp += " leave = '病假' and checkdate %s group by name) ";
```





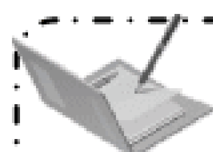
```
        temp += " bjdays on emp.emp_name = bjdays.name";
        temp += " left join";
        temp += " (select isnull(sum(DATEDIFF(second,onleave,offleave)) ";
        temp += " / 60.0 / 60.0 / 8.0,0) as sjday,name    From tab_Check where ";
        temp += " leave = '事假' and checkdate %s group by name) ";
        temp += " sjdays on emp.emp_name = sjdays.name";
        temp += " %s";
        where.Format(" where emp.emp_name = '%s'",m_emp);
        if (m_emp == "(全部)")
            str.Format(temp,datestr,datestr,datestr,datestr,datestr,"");
        else
            str.Format(temp,datestr,datestr,datestr,datestr,datestr,where);
        dataset.Open(str,adLockUnspecified);
        for (int i = 0; i < dataset.GetRecordCount() ; i++)
        {
            int n = 0;
            m_list.InsertItem(i,""); m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["emp_name"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["workday"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["lateday"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["leaveearlyday"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["bjday"]->Value);
            m_list.SetItemText(i,n++,(_bstr_t)dataset.GetFields()->Item["sjday"]->Value);
            dataset.Next();
        }
}
```

（4）向对话框中添加 OnInitDialog 方法，在对话框初始化时设置列表视图控件的表头和列宽度，以及汇总查询条件选择控件的设置，实现代码如下：

```
BOOL CCheckSum::OnInitDialog()
{
        CDialog::OnInitDialog();
        int i = 0;
        m_list.InsertColumn(i,"人员姓名");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"工作总天数");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"迟到总天数");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"早退总天数");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"病假总天数");
        m_list.SetColumnWidth(i++,100);
        m_list.InsertColumn(i,"事假总天数");
        m_list.SetColumnWidth(i++,100);
        m_list.SetExtendedStyle(LVS_EX_FULLROWSELECT|LVS_EX_GRIDLINES);
        int curyear,curmonth;
        CTime time(CTime::GetCurrentTime());
        curyear = time.GetYear();
        curmonth = time.GetMonth();
```


Note

```
        char value[10];
        for (int y = 2000; y < 2100 ;y++)
        {
❶               _itoa(y,value,10);
                m_cyy.InsertString(y-2000,value);
        }
❷       m_cyy.SetCurSel(curyear-2000);
        for (int n = 1 ; n<=12 ;n++)
        {
                _itoa(n,value,10);
                m_cmm.InsertString(n-1,value);
        }
        m_cmm.SetCurSel(curmonth-1);
        CADODataSet dataset;
        dataset.SetConnection(::GetConnection());
        dataset.Open("Select * From tab_Employees");
        m_cemp.InsertString(0,"(全部)");
        for (int index = 1 ; index < dataset.GetRecordCount() ; index++)
        {
                m_cemp.InsertString(index,(_bstr_t)dataset.GetFields()->Item["emp_name"]->Value);
                dataset.Next();
        }
        m_cemp.SetCurSel(0);
        UpdateList();
        return TRUE;
}
```


说明：

❶ _itoa 函数：该函数用于将整型数据转换为字符串类型。

❷ SetCurSel 方法：该方法用于设置组合框中的选中项。

18.12 开发技巧与难点分析

18.12.1 调用动态链接库设计界面

为了使人事考勤管理系统在界面上更加美观，笔者特别重绘了程序界面，并且将其封装成了动态链接库，使读者可以方便地使用。下面就来看一下如何调用动态链接库美化界面。操作步骤如下：

（1）将光盘中提供的 SkinHook.h、SkinLib.dll、SkinLib.lib 文件复制到程序根目录下，如图 18.26 所示。

（2）在工作区窗口选择 FileView 选项卡，右击 Header Files 节点，在弹出的快捷菜单中选





择 Add Files to Folder 命令，在弹出的对话框中找到 SkinHook.h 文件并导入。

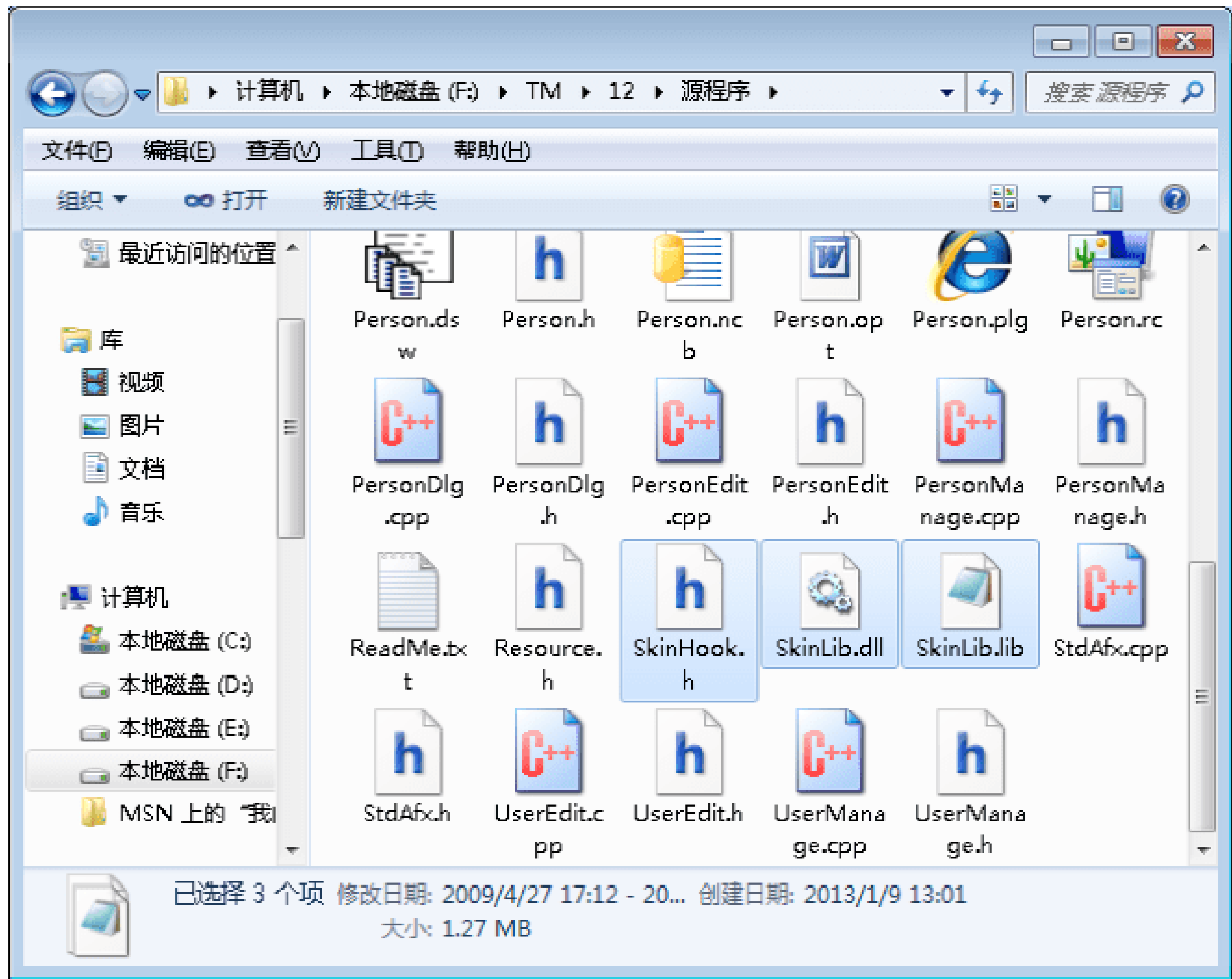


图 18.26　复制动态链接库文件

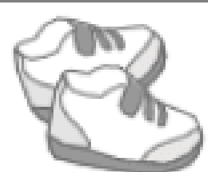
（3）在 Person.cpp 文件中引用 SkinHook.h 文件，并在 InitInstance 方法中调用动态链接库中的 LoadSkin 函数进行界面的绘制。

18.12.2　主窗口的界面显示

在开发本系统的主窗口时，为了使程序看起来更加美观，在程序的背景部分绘制了一幅位图，但是在程序进行最大化时，却出现了问题，程序的背景位图没有被重绘，导致左上角的部分显示一个小图，这要怎么解决呢？

可以先捕获最大化消息，然后调用 Invalidate 函数进行刷新即可。上一个问题的思路有了，可是多了一个新的问题，怎样捕获最大化消息，可以在主窗口的 OnSize 消息处理函数中进行捕获，代码如下：

```
void CPersonDlg::OnSize(UINT nType, int cx, int cy)
{
	CDialog::OnSize(nType, cx, cy);
	if (nType == SIZE_MAXIMIZED)				//捕获最大化消息
	{
		Invalidate();					//刷新
	}
	else if (nType == SIZE_RESTORED)			//捕获还原消息
	{
		Invalidate();					//刷新
	}
}
```



通过上述代码就可以解决调整主窗口大小时背景位图显示不正确的问题了。



Note

18.13 本 章 小 结

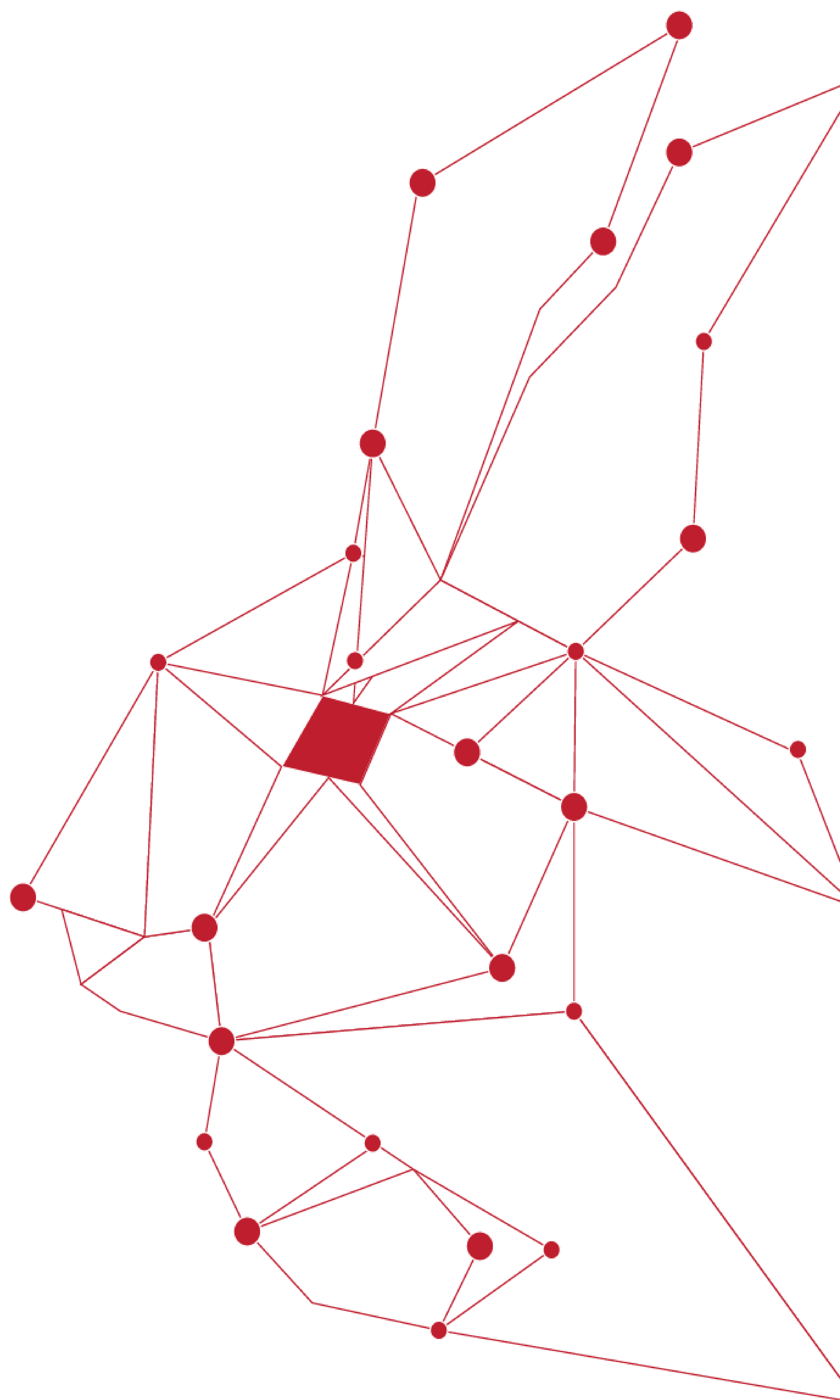
本章通过使用 SQL Server 2008 数据库向读者介绍如何开发人事考勤管理系统。通过本章的学习，读者可以更好地掌握 SQL Server 2008 数据库开发技术，增强对于数据库管理系统开发流程的了解，从而使用户可以向独立开发软件迈出一大步。

人工智能与大数据系列

智能搜索

大数据搜索引擎原理及算法解析

沙　芸　编著





清华大学出版社

人工智能与大数据系列

智能搜索：大数据搜索引擎原理及算法解析

沙　芸　编著

清華大学出版社
北　京

内容简介

本书介绍大数据分布式搜索引擎开发原理与技术实现，主要内容包括多种语言的文本处理、分布式算法与代码实现、Elasticsearch的使用与原理等，通过一个医药领域垂直搜索引擎和电商搜索来说明如何开发实际的大数据智能搜索引擎。全书共分6章，第1章着重介绍开发智能搜索引擎可以采用的软硬件环境；第2～5章着重讨论构建分布式智能搜索引擎可能需要的多种语言文本处理方法，例如Kaldi语音识别实现和基于Raft共识协议的分布式计算平台实现；第6章介绍医药和电商搜索两个应用案例。

本书适合作为高等院校计算机、软件工程专业本科生、研究生的参考用书，对于对人工智能领域感兴趣的人士也有一定的参考价值。

图书在版编目（CIP）数据

智能搜索：大数据搜索引擎原理及算法解析/沙芸编著. —北京：清华大学出版社，2019
（人工智能与大数据系列）
ISBN 978-7-302-53550-8

Ⅰ. ①智…　Ⅱ. ①沙…　Ⅲ. ①搜索引擎—程序设计　Ⅳ. ①TP391.3

中国版本图书馆CIP数据核字（2019）第180067号

责任编辑： 张　敏
封面设计： 常雪影
责任校对： 胡伟民
责任印制： 杨　艳

出版发行： 清华大学出版社
网　　址： http：//www.tup.com.cn, http：//www.wqbook.com
地　　址： 北京清华大学学研大厦A座　**邮　　编：** 100084
社 总 机： 010-62770175　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈： 010-62772015, zhiliang@tup.tsinghua.edu.cn
印 装 者： 北京嘉实印刷有限公司
经　　销： 全国新华书店
开　　本： 186mm×240mm　**印　　张：** 13　**字　　数：** 320千字
版　　次： 2019年11月第1版　**印　　次：** 2019年11月第1次印刷
定　　价： 69.80元

产品编号： 081028-01

前言

PREFACE

智慧生物与机器集群构建的搜索系统已进化成为强大的智能系统。搜索引擎服务早已成为人们生活中不可或缺的一部分。

搜索引擎技术有着悠久的发展历史。1990 年以来，搜索引擎经历了从 Archie 那样的 FTP 文件搜索服务到谷歌网页搜索服务的转变。强化学习、深度学习等技术的发展为搜索引擎技术持续不断地注入新的活力。

本书共分 6 章：第 1 章介绍开发智能搜索引擎可以采用的软件和硬件基础；第 2 章介绍搜索引擎理解文本语义的一些方法；第 3 章介绍通过开发语音识别技术来索引音频信息的一种方法；第 4 章介绍使用 Elasticsearch 实现的大数据分布式搜索引擎；第 5 章介绍分布式计算平台中的共识算法和远程过程调用（RPC）框架；第 6 章介绍医药垂直搜索引擎和电商搜索的案例分析。

本书相关的参考软件和代码在读者 QQ 群（661922108）的附件中可以找到。一些具体的细节也可以在读者 QQ 群讨论。感谢早期合著者、合作伙伴、员工、学员、读者的支持。他们的支持给我们提供了良好的工作基础，就像玻璃容器中的水培植物一样，这是一个持久可用的成长基础。技术的融合与创新无止境，欢迎一起探索。

本书适合需要具体实现搜索引擎的程序员使用，对于信息检索等相关领域的研究人员也有一定的参考价值，同时猎兔搜索技术团队已经开发出以本书为基础的专门培训课程和商业软件。

参与本书编写的还有罗刚、张子宪、石天盈、张继红、刘晓波、叶虎、罗庭亮、柳若边，在此一并表示感谢。

作者

CONTENTS

目录

第 1 章 智能搜索引擎开发

本章首先介绍人工智能的基本知识，然后以商品搜索为例介绍智能搜索引擎。

1.1　人工智能与智能搜索引擎

在计算机科学中，人工智能（artificial intelligence，AI），有时也称为机器智能，是机器展示的智能，与人类和其他动物展示的自然智能形成鲜明对比。计算机科学将人工智能研究定义为对“智能代理”的研究：任何能够感知其环境并采取最大化其成功实现目标的机会的设备。更详细的是，Kaplan 和 Haenlein 将人工智能定义为“系统正确解释外部数据，从这些数据中学习，并通过灵活适应实现特定目标和任务的能力”。通俗地说，当机器模仿人类与其他人类思维相关的“认知”功能时，应用“人工智能”这一术语，如“学习”和“解决问题”。

Kaplan 和 Haenlein 将人工智能分为三种不同类型的人工智能系统：分析人工智能、人类启发人工智能和人性化人工智能。分析人工智能只具有与认知智能相一致的特征，从而产生世界的认知表征，并使用基于过去经验的学习来为未来的决策提供信息。除了认知元素之外，人类启发人工智能还具有认知和情感智能，理解并在决策中考虑人类情感。人性化人工智能显示了所有类型能力（即认知、情感和社交智能）的特征，能够在与他人的交互中体现自我意识。

人工智能研究的传统问题（或目标）包括推理、知识表示、计划、学习、自然语言

处理、感知，以及移动和操纵物体的能力。通用智能属于该领域的长期目标。其方法包括统计方法、计算智能和传统的符号人工智能。人工智能中使用了许多工具，包括搜索和数学优化的版本、人工神经网络，以及基于统计学、概率论和经济学的方法。人工智能领域涉及计算机科学、信息工程、数学、心理学、语言学、哲学等。

人工智能领域的基础是人类智能“可以如此精确地描述，可以使机器模拟它”。这提出了关于心灵本质的哲学论证以及创造具有人类智慧的人造生物的伦理。这些问题是自古以来神话、小说和哲学所探讨的问题。有些人认为，如果人工智能进展有增无减，就会对人类构成威胁；也有些人认为，与以往的技术革命不同，人工智能会造成大规模失业的风险。

21 世纪，人工智能技术在计算机能力、大量数据和理论理解的同步发展之后经历了复苏；人工智能技术已经成为技术行业的重要组成部分，有助于解决计算机科学、软件工程和运筹学中的许多挑战性问题。

强化学习（reinforcement learning）这一领域研究人工（和自然）系统如何学习在复杂环境中基于外部和可能延迟的反馈作出决策。强化学习是人工智能的一个重要分支，它也是 DeepMind 公司的围棋软件阿尔法狗这样的智能系统的重要组成部分。

如果把搜索引擎看作智能体（agent），把用户看作环境（environment），则商品的搜索问题可以被视为典型的顺序决策问题（sequential decision making problem）：

（1）用户每次请求访问页面时，智能体作出相应的排序决策，将商品展示给用户。

（2）用户根据智能体的排序结果，给出点击、翻页等反馈信号。

（3）智能体接收反馈信号，在新的页面访问请求时作出新的排序决策。

（4）这样的过程将一直持续下去，直到用户购买商品或者退出搜索。

在以上问题的形式化中，智能体每一次策略的选择可以被看作一次试错。在这种反复不断试错的过程中，智能体将逐步学习到最优的排序策略。而这种在与环境交互的过程中进行试错的学习，正是强化学习的根本思想。

1.2 Linux 操作系统基础

很多大数据搜索引擎运行在 Linux 操作系统中。Linux 来源于 UNIX，Linux 是 UNIX 操作系统的开放源代码实现。

1.2.1 SSH 远程登录

通过 SSH 客户端软件可以连接到远程的 Linux 服务器。SSH 服务器通常作为大多

数 Linux 发行版上易于安装的软件包提供。用户可以尝试使用 ssh localhost 来测试它是否正在运行。

如果有现成的 Linux 服务器可用，可以使用支持 SSH 协议的终端仿真程序 SecureCRT 连接到远程 Linux 服务器。因为可以保存登录密码，所以比较方便。除了 SecureCRT，还可以使用开源软件 PuTTY(http://www.chiark.greenend.org.uk/~sgtatham/putty)，或者使用可以保存登录密码的 KiTTY(https://www.fosshub.com/KiTTY.html)。如果是用 root 账户登录，则终端提示符是#，否则终端提示符是$。

也可以在 Windows 操作系统下安装 Cygwin，使用它来练习 Linux 常用命令。

使用 VMware，Linux 可以运行在 Windows 系统下。VMware 让 Linux 运行在虚拟机中，而且不会破坏原来的 Windows 操作系统。首先要准备好 VMware，当然仍然需要 Linux 光盘文件。

就好像华山派有剑宗和气宗，Linux 也有很多种版本，例如 RedHat 或者 Ubuntu，以及 SUSE 等。这里介绍 Ubuntu(https://www.ubuntu.com)和 CentOS(http://www.centos.org/)。

操作系统中可能会安装好几个版本的 JDK，在 Linux 中，为了切换 JDK 版本，只需要修改/etc/alternatives 中的符号链接指向。

在 Ubuntu 中，如果需要安装软件，可以下载 DEB 格式的安装包，然后使用 dpkg 命令安装。但一个软件包可能依赖其他软件包。为了安装一个软件可能需要下载其他几个它所依赖的软件包。

为了简化安装操作，可以使用高级包装工具（Advanced Packaging Tool，APT）。APT 会自动计算出程序之间的相互关联性，并且计算出完成软件包的安装需要哪些步骤。这样在安装软件时，不会再被那些关联性问题所困扰。

在/etc/apt/sources.list 文件中指示了包的来源的存储库。包的来源可以是 CD 或 DVD，硬盘上的目录或 HTTP 或 FTP 服务器上的目录。请求的数据包位于服务器（或本地硬盘）上，它将自动下载并安装。APT 主要关注采购包，包的可用版本的比较以及包档案的管理。实际上，可以通过浏览器浏览在 FTP 或 HTTP 上的存储库。

如果需要修改/etc/apt/sources.list 文件，可以先备份这个文件：

```
# sudo cp /etc/apt/sources.list /etc/apt/sources.list.bak
```

如果这一步出现：

```
sudo: unable to resolve host t-000004
```

这样的错误，则可以考虑执行如下的命令修改/etc/hosts 文件的内容：

```
# echo $(hostname -I | cut -d\  -f1) $(hostname) | sudo tee -a /etc/hosts
```

如果安装过程中出现“E: Could not get lock /var/lib/dpkg/lock”这样的错误，则可以尝试使用如下命令修复：

```
# sudo fuser -cuk /var/lib/dpkg/lock
# sudo rm -f /var/lib/dpkg/lock
```

在 CentOS 中，如果需要安装软件，可以下载 RPM 安装包，然后使用 RPM 安装包进行安装。例如，下载 Elasticsearch 软件的安装包 elasticsearch-6.6.0.rpm：

```
# wget https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-6.6.0.rpm
```

使用如下命令安装：

```
# rpm -ivh elasticsearch-6.6.0.rpm
```

但有些操作系统对应的 RPM 安装包找起来往往比较麻烦。

为了简化安装操作，可以使用黄狗升级管理器（Yellow dog Updater,Modified），一般简称 YUM。YUM 会自动计算出程序之间的相互关联性，并且计算出完成软件包的安装需要哪些步骤。

YUM 软件包管理器自动从网络下载并安装软件。YUM 有点类似 360 软件管家，但是不会有商业倾向的推销软件。例如安装支持 wget 命令的软件：

```
#yum install wget
```

1.2.2 Micro 文本编辑器

为了方便在服务器端开发 Python/Perl 相关应用，可以采用 Micro(https://github.com/zyedidia/micro)这样的终端文本编辑器。

在 Linux 上，可以通过 snap 安装 micro：

```
# snap install micro --classic
```

可以使用它编辑配置文件：

```
#./micro  run.pl
```

输入：

```
die "run.pl: Hello Error";
```

这里的 die 表示终止脚本运行，并显示出 die 后面的双引号里面的内容。

保存文件后，使用 Ctrl+q 组合键退出。

1.2.3 Linux Shell 脚本基础

Shell 是用户和 Linux 内核之间的接口程序。用户在命令行提示符下输入的每个命令都由 Shell 先解释再传给 Linux 内核。

Shell 是一个命令语言解释器，拥有自己内建的 Shell 命令集。此外，Shell 也能被系统中其他有效的 Linux 实用程序和应用程序所调用。

Shell 具有如下主要功能。

- 命令解释功能：将用户可读的命令转换成计算机可理解的命令，并控制命令执行。
- 输入输出重定向：操作系统将键盘作为标准输入、显示器作为标准输出。当这些定向不能满足用户需求时，用户可以在命令中用符号“>”或“<”重新定向。
- 管道处理：利用管道将一个命令的输出送入另一个命令，实现多个命令组合完成复杂命令的功能。
- 系统环境设置：用 Shell 命令设置环境变量，维护用户的工作环境。
- 程序设计语言：Shell 命令本身可以作为程序设计语言，将多个 Shell 命令组合起来，编写能实现系统或用户所需功能的程序。

有很多种 Shell，例如 zshell 和 fish。目前一般使用 Bash 脚本。

1.2.4　Shell 脚本

在屏幕上输出“Hello”：

```
echo "Hello"
```

将 ABC 分配给 a：

```
a=ABC
```

输出 a 的值：

```
echo $a
```

将 ABC.log 分配给 b：

```
b=$a.log
```

输出 b 的值：

```
# echo $b
ABC.log
```

把文件“ABC.log”的内容写入到 testfile

```
cat $b > testfile
```

指令“--help”会输出帮助信息。

可以把重复执行的 Shell 脚本写入到一个文本文件。在 Linux 中，文件扩展名不作为系统识别文件类型的依据，但是可以作为用户识别文件的依据，可以简单地将脚本文件以.sh 作为扩展名。

在 Linux 下，可以通过 vi 命令创建一个诸如 script.sh 的文件：vi script.sh。创建好脚本文件后就可以在文件内用脚本语言要求的格式编写脚本程序了。

在创建的脚本文件中输入以下代码并保存退出：

```
#! /bin/bash
echo "hello world!"
```

添加脚本文件的可执行运行权限 chmod 777 script.sh 后，运行文件./script.sh 得到结果：

```
hello world!
```

Shell 脚本中用#表示注释，相当于 C 语言的//注释。但如果#位于第一行开头，并且是#!（称为 Shebang）则例外，它表示该脚本使用后面指定的解释器/bin/sh 解释执行。每个脚本程序必须在开头包含这个语句。

使用参数 n 检查语法错误，例如：

```
# bash -n ./test.sh
```

如果 Shell 脚本有语法错误，则会提示错误所在行；否则，不输出任何信息。

if 语句的语法是：

```
if [ condition ] then
      command1
elif   # 和 else if 等价
    then
         command2
    else
       default-command
fi
```

这里的 fi 就是 if 反过来写。

例如，为了判断某个命令是否存在，可以使用如下格式：

```
if which programname >/dev/null; then
    echo exists
else
    echo does not exist
fi
```

判断 yum 是否存在的例子：

```
if which yum >/dev/null; then
    echo "exists"
else
    echo "does not exist"
fi
```

case 语句的语法是：

```
case 字符串 in
    模式 1)
        语句
        ;;
    模式 2)
         语句
```

```
        ;;
    *)
        默认执行的 语句
        ;;
esac
```

这里的 esac 就是 case 反过来写。例如：

```
extension="png"
case "$extension" in
  "jpg"|"jpeg")
    echo "It's image with jpeg extension."
  ;;
  "png")
    echo "It's image with png extension."
  ;;
  "gif")
    echo "Oh, it's a giphy!"
  ;;
  *)
    echo "Woops! It's not image!"
  ;;
esac
```

这里使用 “|” 把"jpg"和"jpeg"这两个模式连接到了一起。

介绍 4 种模式匹配：

- ${variable#pattern}

从$string 的前面删除$substring 的最短匹配

- ${variable##pattern}

从$string 的前面删除$substring 的最长匹配

- ${variable%pattern}

从$string 的后面删除$substring 的最短匹配

- ${variable%%pattern}

从$string 的后面删除$substring 的最长匹配

使用模式匹配的例子：

```
x=/home/cam/book/long.file.name
echo ${x#/*/}
echo ${x##/*/}
echo ${x%.*}
echo ${x%%.*}
   cam/book/long.file.name
   long.file.name
   /home/cam/book/long.file
   /home/cam/book/long
```

安装 fish Shell：

```
# sudo apt-get install fish
```

1.2.5 AWK

典型的 AWK 程序充当过滤器。它从标准输入读取数据，并输出标准输出的过滤数据。它一次读取数据的一个记录。默认情况下，一次读取一行文本。每次读取记录时，AWK 自动将记录分隔到字段中。字段在默认情况下也是由空格分隔的。每个字段被分配给一个变量，该变量有一个数字名称。变量$ 1 是第一个字段，$ 2 是第二个字段，以此类推。$ 0 表示整个记录。此外，还设置了一个名为 NF 的变量，其中包含在记录中检测到的字段的数目。

来试试一个很简单的例子。过滤 ls 命令的输出：

```
# ls -l ./ | awk '{print $0}'
```

显示文本文件 nohup.out 匹配（含有）字符串"sun"的所有行。

```
# awk '/sun/{print}' nohup.out
```

由于显示整个记录（全行）是 awk 的默认动作，因此可以省略 action 项。

例如，得到 Python 的版本号：

```
# python 2>&1 --version | awk '{print $2}'
2.7.5
```

这里 2>&1 的意思是：把标准错误重定向到标准输出。

1.3 Java 基础

在 Ubuntu 下安装 JDK 可以参考：

https://github.com/AdamScheller/UbuntuJavaInstaller/blob/master/ujavainstaller.sh

首先从 https://www.oracle.com/technetwork/java/javase/downloads/

jdk11-downloads-5066655.html 下载 jdk-11.0.2_linux-x64_bin.tar.gz；

然后设置环境变量：

```
#JDK_VERSION=jdk-11.0.2
#mkdir -p /usr/local/java
#JDK_ARCHIVE=jdk-11.0.2_linux-x64_bin.tar.gz
#JDK_LOCATION=/usr/local/java/$JDK_VERSION
#tar -xf $JDK_ARCHIVE -C /usr/local/java
#cat >> /etc/profile <<EOF
JAVA_HOME=$JDK_LOCATION
JRE_HOME=$JDK_LOCATION/jre
PATH=$PATH:$JDK_LOCATION/bin:$JDK_LOCATION/jre/bin
```

```
    export JAVA HOME
    export JRE HOME
    export PATH
    EOF
```

增加 Java 需要的软链接：

```
    #update-alternatives --install "/usr/bin/java" "java" "$JDK LOCATION/bin/
java" 1 >> /dev/null
    #update-alternatives --install "/usr/bin/javac" "javac" "$JDK LOCATION/bin/
javac" 1 >> /dev/null
```

检查 Java 是否正确安装：

```
    #java
    #javac
```

或者使用源安装：

```
    #sudo add-apt-repository ppa:linuxuprising/java
    #sudo apt install oracle-java11-installer
```

如果要构建源代码工程，可以使用工具 Ant、Maven 或者 Gradle。Ant 与 Maven 都和项目管理软件 make 类似。虽然 Maven 正在逐步替代 Ant，但目前仍然有很多开源项目在继续使用 Ant。

1.3.1 使用 Ant

在 CentOS 下安装 Ant。

```
    # yum -y install ant
```

从 http://ant.apache.org/bindownload.cgi 可以下载到 Ant 的最新版本。

在 Windows 操作系统下，ant.bat 和 ANT_HOME、CLASSPATH、JAVA_HOME 三个环境变量相关。需要用路径设置 ANT_HOME 和 JAVA_HOME 环境变量，并且路径不要以\或/结束，不要设置 CLASSPATH。使用 echo 命令检查 ANT_HOME 环境变量：

```
    >echo %ANT_HOME%
    D:\apache-ant-1.7.1
```

如果把 Ant 解压到 C:\apache-ant-1.7.1，则修改环境变量 PATH，增加当前路径 C:\ apache-ant-1.7.1\bin。

如果一个项目的源代码根路径包括一个 build.xml 文件，则说明这个项目可能是用 Ant 构建的。大部分用 Ant 构建的项目只需要如下一个命令：

```
    #ant
```

集成开发环境 Eclipse 中已经集成了 Ant 组件，所以如果要在 Eclipse 中使用 Ant，则并不需要专门安装 Ant 软件工具。

在 Eclipse 中，可以根据项目源代码自动生成 build.xml 文件。方法是：选定指定 Java

项目，右击菜单中的“导出”选项，选择导出“Ant Build 文件”。

项目的源代码根路径包括一个 build.xml 文件，每一个 build.xml 只有一个 Project，Project 表示一个工程，里面定义了这个工程的全局属性。

执行 Ant 时，可以选择执行哪个目标。当没有指定目标时，执行 project 的 default 属性所确定的目标。只需要如下一个命令执行默认的目标：

```
#ant
```

build.xml 文件定义了一个项目。项目相关的信息包括项目名称和默认编译的目标。例如项目 KaldiJava 默认编译的目标是 makeJAR：

```
<project name="kaldi" default="makeJAR" basedir=".">
```

可以运行指定的任务，例如运行下面的 compile 任务：

```
<target name="compile" depends="init"
  description="compile the source ">
  <javac compiler="modern" encoding="utf-8" debug="true" srcdir="${src}"
destdir="${bin}" classpathref="project.class.path" target="1.8" source="1.8" />
</target>
```

使用命令行：

```
#ant compile
```

通过 Ant 执行的 build.xml 文件来自动生成可执行的 jar 包。Ant 通过调用目标树，就可以执行各种目标。例如，编译源代码的目标，还有打 jar 包的目标。

由于 Ant 构建的文件是 XML 格式的文件，所以很容易维护和书写，而且结构很清晰。Ant 可以集成到开发环境中。Eclipse 默认安装了 Ant 插件。选中 build.xml 文件后，在 run as 中选取 ant build，就可以运行 build.xml 文件中的默认目标了。

使用 build.xml 文件可以做的事情有：

- 定义全局变量，例如定义项目名。
- 初始化，主要是建立目录，例如发布路径。
- Java 源代码编译成为 class 文件；调用<javac encoding="utf-8" debug="true" srcdir="${src}" destdir="${bin}" classpathref="project.class.path" target="1.6" source= "1.6"/>。
- 把 class 文件打包到一个 jar 文件；调用<jar destfile="***">。
- 建立 API 文档。

目标之间可以有依赖关系。例如，makeJAR 依赖 init 和 compile，init 依赖 clean，所以目标执行顺序是 clean→init→compile→makeJAR。

```
<target name="makeJAR" depends="init,compile">
```

如果需要更新 war 中的文件，就设置 update="true"。

jar 包中要正好包含有用的 class 文件，既不能包含测试部分代码，也不能包含源文件。

```
<target name="makeJAR" depends="init,compile">
   <jar destfile="${dist}/${jarfile}">
    <fileset dir="${bin}">
      <include name="**/*.class"/>
      <exclude name="**/*.jflex"/>
     </fileset>
   </jar>
</target>
```

javac 标签调用 javac 编译器。如果 Java 源代码文件编码不一致可能会出错，可以把编码统一成 GBK 或者 UTF-8。如果源代码文件编码是 UTF-8，则使用 javac 编译时，要增加 encoding 选项指定编码是 UTF-8。

```
<javac  encoding="utf-8"  debug="true"  srcdir="${src}"  destdir="${bin}"
classpathref="project.class.path" target="1.6" source="1.6"/>
```

可以在 junit 任务中使用 batchtest：

```
<target name="test" depends="compileTest">
  <junit>
    <classpath>
      <pathelement location="bin" />
      <pathelement location="lib/junit-4.10.jar"/>
    </classpath>
    <batchtest>
      <fileset dir="${test}">
          <include name="**/*Test*" />
      </fileset>
    </batchtest>
    <formatter type="brief" usefile="false"/>
  </junit>
</target>
```

从命令行运行测试。

```
#ant test
```

1.3.2　使用 Maven

可以从 http://maven.apache.org/download.html 下载最新版本的 Maven，当前版本是 Maven-3.3.9。解压下载的 Maven 压缩文件到 C:根路径，将创建一个 C:\apache-maven-3.3.9 路径。修改 Windows 系统环境变量 PATH，增加当前路径 C:\apache-maven-3.3.9\ bin。如果一个项目的源代码根路径包括一个 pom.xml 文件，则说明这个项目可能是用 Maven 构建的。大部分用 Maven 构建的项目只需要如下一个命令：

```
#mvn clean install
```

可以在 PowerShell 下使用 scoop 命令安装 Maven：

```
>scoop install maven
```

在项目的根目录中放置 pom.xml，在 src/main/java 目录中放置项目的运行代码，在 src/test/java 中放置项目的测试代码。

使用 Maven Archetype 来创建项目的结构。采用 Maven 构建的项目一般包括一个 pom.xml 文件。

```
<build>
    <plugins>
        <plugin>
            <artifactId>maven-assembly-plugin</artifactId>
            <configuration>
                <archive>
                    <manifest>
                        <mainClass>fully.qualified.MainClass</mainClass>
                    </manifest>
                </archive>
                <descriptorRefs>
                    <descriptorRef>jar-with-dependencies</descriptorRef>
                </descriptorRefs>
            </configuration>
        </plugin>
    </plugins>
</build>
```

使用下面的命令执行它：

```
mvn assembly:single
```

用 install 参数下载依赖的 jar 文件。

```
mvn install
```

Maven 默认的本地仓库地址为${user.home}/.m2/repository。例如，如果用 Administrator 账户登录，则把 jar 包下载到 C:\Users\Administrator\.m2\repository\这样的路径。

如果 jar 文件位于 lib 路径下，则 Eclipse 的.classpath 文件中的 classpathentry 是 lib 类型。

```
<classpathentry kind="lib" path="lib/commons-io-1.2.jar"/>
```

如果 jar 包位于 Maven 的存储库中，则 Eclipse 的.classpath 文件中的 classpathentry 是 var 类型。

```
<classpathentry kind="var" path="M2_REPO/junit/junit/4.8.2/junit-4.8.2.jar"
            sourcepath="M2_REPO/junit/junit/4.8.2/junit-4.8.2-sources.
jar"/>
```

建大楼的时候需要搭建最终不会交付使用的脚手架。类似地，很多单元测试代码也不会在正式环境中运行，但是必须写。与此类似，可以使用 JUnit 做单元测试。

maven-surefire-plugin 是 Maven 中执行测试用例的插件。从版本 2.22.0 开始，Maven Surefire 和 Maven Failsafe 插件为在 JUnit 平台上执行测试提供本地支持。

POM 文件的构建部分如下：

```
<build>
   <plugins>
      <plugin>
         <groupId>org.apache.maven.plugins</groupId>
         <artifactId>maven-surefire-plugin</artifactId>
         <version>2.22.1</version>
      </plugin>
   </plugins>
</build>
```

如果想使用 Maven Surefire 插件的原生 JUnit 5 支持，则必须确保从类路径中找到至少一个测试引擎实现。所以，在配置 Maven 构建的依赖项时将 junit-jupiter-engine 依赖项添加到测试范围。

```
<dependencies>
   <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-api</artifactId>
      <version>5.4.0</version>
      <scope>test</scope>
   </dependency>
   <dependency>
      <groupId>org.junit.jupiter</groupId>
      <artifactId>junit-jupiter-engine</artifactId>
      <version>5.4.0</version>
      <scope>test</scope>
   </dependency>
</dependencies>
```

如果使用 Maven Surefire 插件的默认配置，它会运行从测试类中找到的所有测试方法，如果测试类的名称为：

- 以字符串 Test 开头或结尾。
- 以字符串 Tests 结尾。
- 以字符串 TestCase 结尾。

1.3.3　使用 Gradle

相比 Maven，Gradle 提供了更加灵活的构建方式。

可以下载二进制文件来安装 Gradle：

```
# cd /opt/
# wget -bc https://services.gradle.org/distributions/gradle-5.2-bin.zip
# unzip gradle-5.2-bin.zip
# mv gradle-5.2 ./gradle
```

可在/etc/profile 文件或/etc/profile.d 目录设置环境变量。不过/etc/profile.d/比/etc/profile 好维护，对于不需要软件的变量，可以直接删除/etc/profile.d/下对应的 shell 脚本。

创建设置环境变量的脚本文件：

```
# echo 'export GRADLE_HOME=/opt/gradle' > /etc/profile.d/gradle.sh
# echo 'export PATH=$PATH:$GRADLE_HOME/bin'  >> /etc/profile.d/gradle.sh
```

执行这个脚本文件：

```
# . /etc/profile.d/gradle.sh
```

检查 gradle 的环境变量是否设置正确：

```
# gradle -v
```

这里，只需要使用 gradle 编译代码。

```
# gradle build
```

如果需要从本地目录中获取 jar 文件，则在 build.gradle 文件中添加：

```
repositories {
  flatDir {
     dirs 'libs'
  }
}

dependencies {
  compile name: 'commons-exec-1.3'
}
```

如果需要把依赖包下载到本地目录，则可以在 build.gradle 文件中增加如下任务：

```
task copyDependencies(type: Copy) {
  from configurations.compile
  into 'lib'
}
```

build.gradle 中通常有两个存储库和依赖项部分。其中一个完全包含在 buildscript {} 部分中。buildscript 部分的依赖项和用于查找它们的存储库仅适用于 build.gradle 脚本本身的代码。这些依赖项通常是 gradle 插件，有时是用于自定义构建代码的依赖项。

例如：一个使用 Spring Boot Gradle 插件的 build.gradle 内容如下：

```
buildscript {
   ext {
      springBootVersion = '2.1.2.RELEASE'
   }
   repositories {
      maven {url 'http://maven.aliyun.com/nexus/content/groups/public/'}
      mavenCentral()
   }
   dependencies {
      classpath("org.springframework.boot:spring-boot-gradle-plugin:
${springBootVersion}")
   }
}

apply plugin: 'java'
```

```
apply plugin: 'eclipse-wtp'
apply plugin: 'org.springframework.boot'
apply plugin: 'io.spring.dependency-management'

group = 'com.lietu'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
    maven {url 'http://maven.aliyun.com/nexus/content/groups/public/'}
   mavenCentral()
}

configurations {
    providedRuntime
}

dependencies {
   implementation 'org.springframework.boot:spring-boot-starter-freemarker'
   implementation 'org.springframework.boot:spring-boot-starter-web'
   providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'

   testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testCompile group: 'junit', name: 'junit', version: '4.11'
}
```

要创建一个 Java 项目，先创建一个新的项目目录，进入并执行：

```
# gradle init --type java-library
```

会得到源文件夹和 Gradle 构建文件。

如果使用默认 gradle 包装结构建立项目，则执行：

```
src/main/java
src/main/resources
src/test/java
src/test/resources
```

如果需要将已有的 Maven 项目转换为 Gradle 项目，则可以在项目根目录下运行如下命令来生成 build.gradle 文件：

```
#gradle init
```

不需要修改 sourceSets 来运行测试。Gradle 会发现测试类和资源都在 src/test 中。然后可以运行：

```
#./gradlew test
```

可以编写测试类，然后使用 Gradle 运行一个测试用例：

```
# gradle test -Dtests.class=LibraryTest
# gradle test "-Dtests.class=*.ClassName"
```

运行包和子包中所有的测试用例：

```
# gradle test "-Dtests.class= com.lietu.package.*"
```

Gradle 构建项目的默认行为是执行测试。在构建时，也可以不执行测试：

```
# gradle build -x test
```

大多数工具需要在计算机上进行安装才能使用。这时构建用户可能会造成不必要的负担。同样重要的是，用户是否会为构建工具安装正确版本的工具？如果用户正在旧版本上构建软件怎么办？

Gradle Wrapper（以下称为 Wrapper）解决了这两个问题，是开始 Gradle 构建的首选方式。使用 Wrapper 可以使项目组成员不必预先安装好 Gradle，以便于统一项目所使用的 Gradle 版本。如果需要，也可以手工修改 gradle-wrapper.properties 文件中的 distributionUrl 值以改变 Gradle 版本。

在 Windows 操作系统下可以添加环境变量 GRADLE_USER_HOME 自定义 Gradle 缓存位置。

为了安装 Gradle 的 Eclipse 插件，可以从 https://github.com/eclipse/buildship 找到安装地址。

利用 Eclipse→Window→Preferences→Gradle 菜单命令设定 Gradle User Home 路径 F:\ soft\gradle-5.2\bin。

1.3.4 使用 Groovy Shell

Groovy Shell 是一个命令行应用程序，可以使用它来评估 Groovy 表达式、函数，定义类和运行 Groovy 命令。

在 Ubuntu 下使用 SDKMAN!(https://sdkman.io)安装 Groovy Shell。首先安装 zip：

```
# apt install zip
```

然后安装 SDKMAN!。首先下载并运行安装脚本：

```
# curl -s "https://get.sdkman.io" | bash
```

在终端上输入以下代码段：

```
# source "$HOME/.sdkman/bin/sdkman-init.sh"
```

然后安装最新的稳定 Groovy 版本：

```
# sdk install groovy
```

安装完成后，使用以下命令进行测试：

```
# groovy -version
```

运行 Groovy Shell：

```
# groovysh
groovy:000> print("hi")
```

1.3.5　使用 JShell

Java 有 BeanShell(https://github.com/beanshell/beanshell)这样的第三方 REPL 工具。但是随着 Java 9 于 2016 年发布，SDK 开始附带一个名为 JShell 的官方 REPL。

```
C:\test>jshell
|  欢迎使用 JShell -- 版本 11.0.1
|  要大致了解该版本，请输入：/help intro
```

用户可以在 Jshell 中执行算术运算，例如：

```
jshell> 10+15
$1 ==> 25
```

执行省略了分号的 Java 语言语句：

```
jshell> System.out.print("hello")
hello
```

加载 jar 包：

```
jshell> /env -class-path C:\lib\guava-19.0.jar
|  正在设置新选项并还原状态。
```

可以编写 JShell 脚本，然后调用 jshell 命令来执行。

1.4　Python 基础

首先介绍如何在 Windows 下和 Linux 下安装 Python。

1.4.1　Windows 下安装 Python

使用 Chocolatey 安装 Python。在 PowerShell 提示符下输入：

```
>choco install python3
```

使用如下命令检查 Python 是否正确安装，以及所使用的版本号：

```
> python --version
```

1.4.2　Linux 下安装 Python

首先检查 Python 3 是否已经正确安装，以及所使用的版本号：

```
# python3 -V
Python 3.4.5
```

检查 Python 3 所在的路径：

```
# which python3
/usr/bin/python3
```

如果使用 CentOS，可以使用 yum 安装 Python 3。首先查找可供安装的 Python 版本：

```
# yum search python3
```

然后安装想要的版本：

```
# yum install python36
```

如果使用 Ubuntu，则可以首先添加 PPA 软件源。

```
$ sudo add-apt-repository ppa:deadsnakes/ppa
```

然后更新软件源。

```
$ sudo apt update
```

最后执行安装：

```
$ sudo apt install python3.7
```

1.4.3　开发环境

简单地，可以使用 Nodepad++这样的文本编辑器写 Python 代码，也可以使用 PyDev 或者 PyCharm 集成开发环境。有两个版本的 PyCharm：专业版（免费 30 天试用版）和功能较少的社区版本（Apache 2.0 许可证）。

这里介绍使用 PyDev(http://www.pydev.org)。PyDev 是 Eclipse 的第三方插件。它是一个集成开发环境（IDE），用于 Python 编程，支持代码重构、图形调试、代码分析等功能。

PyDev 要求用户首先安装 Python 解释器和 Eclipse 集成开发环境。PyDev 是 Eclipse 的一个插件，如果没有安装 Python 和 Eclipse，就无法使用它。

首先要准备好 JDK 和集成开发环境 Eclipse。当前可以使用 JDK11。JDK11 可以从 Java 官方网站 http://www.oracle.com/technetwork/ java/index.html 下载得到。使用默认方式安装即可。

Eclipse 默认是英文界面，如果习惯用中文界面可以从 http://www.eclipse.org/babel/downloads.php 下载支持中文的语言包。

如果想要切换回英文开发环境，则可以使用命令行进入 Eclipse 主目录后输入：

```
eclipse -nl en
```

切换回简体中文：

```
eclipse -nl zh_CN
```

然后安装插件，配置路径。

插件通过 Eclipse 的软件安装机制安装。从“帮助”菜单中选择“安装新软件”，输

入网址 http://pydev.org/updates，并选择 PyDev 执行 Next 开始安装，安装完需要重启 Eclipse。

必须配置 PyDev 才能与 Eclipse 和 Python 设置一起正常工作。从 Eclipse 主菜单中选择 Window→Preferences，将打开“首选项”对话框。选择 PyDev→Interpreter – Python，单击 Quick Auto-Config 按钮可以按照用户所希望的方式设置 Eclipse，但前提是选择了某个版本的 Python 3。如果未找到任何版本的 Python，请执行以下步骤：如果找到早期版本的 Python，请确保使用“删除”按钮将其删除，然后手动选择 python.exe 文件所在的路径。

1.5　C#基础

C#是一门贴近自然语言的高级编程语言。因为是在美国发明的，所以其中的关键字使用英语定义。

交互式的编程环境能立刻对使用者做出回应，所以对于学习一门新的编程语言具有很大的帮助。可以在交互模式下使用 C#编译器。为了使用 C# REPL(REPLRead-Eval-Print Loop)，在 Ubuntu 下，目前可以使用 Mono 的 CSharp Shell 体验。

首先安装相关软件包：

```
# apt install mono-csharp-shell
```

然后启动 CSharp Shell：

```
# csharp
```

输入代码：

```
csharp> Console.WriteLine("hi");
```

在 Windows 下可以使用 scriptcs(https://github.com/scriptcs/scriptcs)体验 REPL：

可以使用 Chocolatey 安装 scriptcs：

```
> choco install scriptcs
```

测试是否安装正确：

```
> Console.WriteLine("hi");
```

安装解析 HTML 格式的文档的包：

```
scriptcs -install dcsoup
```

然后运行解析 XML 的测试方法：

```
> scriptcs
```

使用 dcsoup 解析 XML 格式字符串的 C#代码如下：

```
    > using Supremes;
    > using Supremes.Nodes;
    > using Supremes.Parsers;
    > using System;
    >string html = "<?xml version=\"1.0\" encoding=\"UTF-8\"><tests><test><id>
xxx</id><status>xxx</status></test><test><id>xxx</id><status>xxx</status></te
st></tests></xml>";
    > Document doc = Dcsoup.Parse(html, "", Parser.XmlParser);
    > Console.WriteLine(doc.GetElementsByTag("id"));
```

输出结果如下：

```
    <id>
     xxx
    </id>
    <id>
     xxx
    </id>
```

在 CentOS 下可以使用 dotnet-script(https://github.com/filipw/dotnet-script) 从.NET Core 命令行接口运行 C#脚本。

可以使用以下步骤在 CentOS 7 上安装.NET Core：

```
    # yum install centos-release-dotnet
    #yum install rh-dotnet21 -y
```

使用它：

```
    #scl enable rh-dotnet21 bash
```

查看版本信息：

```
    # dotnet --info
```

安装 dotnet-script：

```
    #dotnet tool install -g dotnet-script
```

使用 dotnet-script：

```
    #dotnet-script
```

使用 HtmlAgilityPack 解析 HTML 字符串的 C#代码如下：

```
    > #r "nuget: HtmlAgilityPack"
    > using System;
    > using System.Xml;
    > using HtmlAgilityPack;
    > var html =
           @"<!DOCTYPE html>
    <html>
    <body>
       <h1>This is <b>bold</b> heading</h1>
       <p>This is <u>underlined</u> paragraph</p>
       <h2>This is <i>italic</i> heading</h2>
    </body>
    </html> ";
```

```
> var htmlDoc = new HtmlDocument();
> htmlDoc.LoadHtml(html);

> var htmlBody = htmlDoc.DocumentNode.SelectSingleNode("//body");

> Console.WriteLine(htmlBody.OuterHtml);
```

可以使用轻量级 C#编辑器 RoslynPad(https://roslynpad.net/)开发简单应用。在 RoslynPad 中引用 HtmlAgilityPack 包的代码如下。

```
#r "$NuGet\HtmlAgilityPack\1.9.0\lib\Net45\HtmlAgilityPack.dll"
```

实际项目开发可以选择 Visual Studio。NuGet 是一个为 Microsoft 开发平台设计的免费开源软件包管理器。它使得在 Visual Studio 中安装和更新第三方库和工具更容易。可以在命令行使用 NuGet 安装项目需要的包。例如使用 NuGet 在当前目录下安装测试包 NUnit：

```
PM> Install-Package NUnit
```

可以手动安装需要的包。例如，使用 NuGet 从本地安装 Newtonsoft.Json。从 NuGet 网站 https://www.nuget.org/packages/Newtonsoft.Json/11.0.2 下载安装包 Newtonsoft.Json.11.0.2.nupkg 到 D:\soft 目录。

然后在 Visual Studio 的包管理器命令行安装：

```
PM> Install-Package Newtonsoft.Json -Version 11.0.2  -Source D:\soft
```

Visual Studio 会把相关的依赖信息写入项目 csproj 文件的 PackageReference 部分。

1.6　硬件基础

日常开发所使用的台式机箱用于放置主板及硬盘、光驱、电源，以及 USB 控制器、显卡等。

像 CPU 这样的位于主机箱内的设备通常称为内部设备（简称内设），而位于主机箱之外的设备通常称为外部设备（简称外设，如显示器、键盘、鼠标等）。通常，主机自身（装上软件后）已经是一台能够独立运行的计算机系统，服务器等有专门用途的计算机通常只有主机，没有其他外设。

为了更好地支持 Tensorflow 这样的深度学习环境，可以考虑选用 Intel CPU。

在 Linux 系统中，可以通过查看/proc/cpuinfo 文件来显示 CPU 信息。运行：

```
#cat /proc/cpuinfo
```

示例部分输出如下：

```
processor        : 31
vendor_id        : GenuineIntel
```

```
    cpu family       : 6
    model            : 85
    model name       : Intel(R) Xeon(R) Gold 5120T CPU @ 2.20GHz
    stepping         : 4
    microcode        : 0x1
    cpu MHz          : 2200.000
    cache size       : 16384 KB
    physical id      : 31
    siblings         : 1
    core id          : 0
    cpu cores        : 1
    apicid           : 31
    initial apicid   : 31
    fpu              : yes
    fpu exception    : yes
    cpuid level      : 13
    wp               : yes
    flags            : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
    pat pse36 clflush mmx fxsr sse sse2 ss syscall nx pdpe1gb rdtscp lm constant tsc
     arch perfmon rep good nopl xtopology eagerfpu pni pclmulqdq ssse3 fma cx16
pcid
     sse4 1 sse4 2 x2apic movbe popcnt tsc deadline timer aes xsave avx f16c rdrand
    hypervisor lahf_lm abm 3dnowprefetch invpcid_single kaiser fsgsbase
tsc adjust b
    mi1 hle avx2 smep bmi2 erms invpcid rtm mpx avx512f rdseed adx smap clflushopt c
    lwb avx512cd xsaveopt xsavec xgetbv1 arat
    bugs             : cpu meltdown spectre v1 spectre v2 spec store bypass
    bogomips         : 4400.00
    clflush size     : 64
    cache alignment  : 64
    address sizes    : 46 bits physical, 48 bits virtual
    power management:
```

1.7 本章小结

Ubuntu 是一款基于 Debian 发行版本派生的操作系统，而 CentOS 则是基于 Red Hat 商业版本派生的操作系统。

除了 Visual Studio、C#集成开发环境，还可以选择在 VS Code 中使用 OmniSharp-Roslyn。

第 2 章 搜索引擎理解语义

搜索引擎根据用户输入的关键词或者问题查询文本。为了快速返回结果，搜索引擎往往对待查询的文本建立倒排索引。对于用户输入问题和索引库中的文本理解有助于提升查询结果的准确性。本章首先介绍处理文本的整体流程，然后介绍基于文法的语言模型，以及 N 元模型。

2.1 处理文本

对于要处理的文章，首先可以切分成句子，然后再做分词和词性标注等处理。可以使用 java.text.BreakIterator 把文本分成句子。BreakIterator.getSentenceInstance 返回按标点符号的边界切分句子的实例。BreakIterator 支持多种语言的文本处理。简单切分出中文句子的方法是：

```
String stringToExamine = "这是一种高效富集甲基化 DNA 的方法。在该方法中，可将与 5mC 特异性结合的抗体加入到变性的基因组 DNA 片段中，从而使甲基化的基因组片断免疫沉淀，形成富集。通过与已有 DNA 微芯片技术相结合，从而进行大规模 DNA 甲基化分析。该方法简便，特异性高，适合 DNA 甲基化组学(DNA Methylome)的分析。通过以上论述，不难看到检测甲基化的方法不断涌现，一方面说明其研究难度之大，另一方面也说明种种方法都有其局限性，诸如对酶的依赖，PCR 扩增的问题，芯片数据分析的标准化问题，等等。综上所述，各种表观基因组学技术方法使我们可以绘制出诸如 DNA 甲基化以及组蛋白修饰模式的详细图谱，为我们研究甲基化的生物学功能，以及在肿瘤生成中的作用，以致肿瘤预防、诊断、治疗和预后方面提供更多信息。然而，为了实现这个宏远目标，需要的不仅仅是支助的增加，还有国际同行的密切合作。";
```

```
//根据中文标点符号切分
BreakIterator boundary = BreakIterator.getSentenceInstance(Locale.CHINESE);
//设置要处理的文本
boundary.setText(stringToExamine);
int start = boundary.first(); //开始位置
for (int end = boundary.next(); end != BreakIterator.DONE;
        start = end, end = boundary.next()) {
    //输出子串，也就是一个句子
    System.out.println(stringToExamine.substring(start, end));
}
```

2.2　基于文法的语言模型

可以使用 JSGF(Java Speech Grammar Format)描述的语言模型来匹配标准问答集。JSGF 中的每个文件只定义一个语法。每个语法包含两部分：语法头和语法体。

语法头格式：#JSGF version char-encoding locale;

例如：#JSGF V1.0 ISO8859-5 en;

声明语法头后，需要指定语法名称。语法名称格式如下：

```
grammar grammarName;
```

接下来定义语法体、语法体定义规则。规则定义格式如下：

```
public <ruleName> = ruleExpansion;
```

例如，定义一个名为 greet 的规则：

```
public <greet> = Hello;
```

一个简单的规则扩展可以引用一个或多个符号或规则。

```
public <greet> = Hello;
public <completeGreet> = <greet> World;
```

一个简单的“Hello World”语法文件。

```
#JSGF V1.0;

grammar simpleExample;

public <greet> = Hello;

public <completeGreet> = <greet> World;
```

还可以在语法文件中添加注释。

```
//单行注释

/*多行注释*/
```

```
/**
*文档注释
* @author luogang
*/
```

JSGFKit(https://github.com/ExpandingDev/JSGFKit)是一个 JSGF 语言模型的实现。JSGFKit 使用 Grammar 类作为持有语法规则的主要容器。使用 JSGFKit 的代码如下：

```
Grammar g = new Grammar();
//创建一个 Rule 对象
Rule greetRule = new Rule("greet",
     new RequiredGrouping(new RuleReference("greetWord")),
     new RequiredGrouping(new RuleReference("name")));
//增加一个规则到文法库
g.addRule(greetRule);
g.addRule(new Rule("greetWord",
        new AlternativeSet(new Token("hello"), new Token("hi"))));
g.addRule(new Rule("name",
     new AlternativeSet(new Token("peter"), new Token("john"),
                  new Token("mary"), new Token("anna"))));

String text = g.compileGrammar();
System.out.println(text);
```

输出结果如下：

```
#JSGF V1.0 UTF-8 zh;
grammar default;
public <greet> = (<greetWord>) (<name>);
public <greetWord> = hello | hi;
public <name> = peter | john | mary | anna;
```

2.3　正则表达式查找文本

正则表达式\w 与任何单词字符匹配，包括下画线，而\w+则匹配一个英文单词。可以在文本编辑器 Nodepad++中测试正则表达式匹配。

java.util.regex 包提供了对正则表达式的支持。其中的 Pattern 类代表一个编译后的正则表达式。通过 Matcher 类根据给定的模式查找输入字符串。通过调用 Pattern 对象的 matcher 方法得到一个 Matcher 对象。使用正则表达式提取字符串的例子如下：

```
String example = "This is my small example string which I'm going to use for
pattern matching.";
Pattern pattern = Pattern.compile("\\w+");
Matcher matcher = pattern.matcher(example);
//检查所有的出现
while (matcher.find()) {
   System.out.print("开始位置: " + matcher.start());
```

```
        System.out.print(" 结束位置: " + matcher.end() + " ");
        System.out.println(matcher.group());
    }
```

用正则表达式检查 E-mail 的格式。用这样的正则表达式：

```
\w+@[\w+\.\w+]+
```

可以匹配上 luogang@lietu.com 这样的文本。

还需要“.”和“-”两个字符。例如邮箱 zhangshna.Mr@163.com，在@符号之前还有个点“.”。

检查 E-mail 格式的语句：

```
String mailTo = "abc@sina.com.cn";
System.out.println(mailTo.matches( "[\\w[.-]]+@[\\w[.-]]+\\.[\\w]+" )); //
输出 true
```

正则表达式的原理是有限状态自动机。dk.brics.automaton(http://www.brics.dk/automaton/)包含有限状态自动机的实现。BasicAutomata.makeChar 方法生成接收单个字符的自动机。

```
Automaton a = BasicAutomata.makeChar('W');        //创建一个字符 W 组成的自动机
```

repeat 方法重复多次。例如重复字符 A～Z 至少一次：

```
Automaton a = BasicAutomata.makeCharRange('A', 'Z');
Automaton c = BasicOperations.repeat(a,1);        //指定最少重复次数
System.out.println(BasicOperations.run(c, "WW")); //输出 true
System.out.println(BasicOperations.run(c, "WWW")); //输出 true
```

BasicOperations.concatenate 方法连接两个自动机。例如：

```
Automaton a = BasicAutomata.makeCharRange('A', 'Z');
Automaton b = BasicAutomata.makeChar('@');
Automaton c = BasicOperations.concatenate(a, b);
System.out.println(BasicOperations.run(c, "A@")); //输出 true
System.out.println(BasicOperations.run(c, "AW")); //输出 false
```

使用 Automaton 类匹配数字：

```
Automaton num = Automaton.minimize((new RegExp("[0-9]+")).toAutomaton());
System.out.println(BasicOperations.run(num, "12356756700")); //输出 true
```

很多时候对匹配对象有更多的要求。根据上下文过滤匹配结果要用到环视结构。如果条件位于要提取的信息的后面，则叫作向前看表达式，否则叫作向回看。

一个向回看的表达式从模式开始，直到向回看的表达式结束为止。

```
(?<=X)        X, 按条件 X 向回寻找
```

例如，查找"http://"后面的文本写做：(?<=http://)。完整的例子如下：

```
//查找"http://"后面的文本
Pattern pat = Pattern.compile( "(?<=http://)\\S+" );
```

```
    String str = "The Java2s website can be found at  http://www.java2s.com. There,
you can find some Java examples.";

    Matcher matcher = pat.matcher(str);
    while (matcher.find())
      System.out.println(":" + matcher.group() + ":");
```

下面的例子提取网页中的链接：

```
    String pageContents = "<a href=\"http://www.lietu.com\">猎兔</a>";
    Pattern p = Pattern.compile("<a\\s+href\\s*=\\s*\"?(.*?)[\"|>]",
                        Pattern.CASE_INSENSITIVE); //忽略大小写
    Matcher m = p.matcher(pageContents);
    while (m.find()) {//打印网页中所有的链接
       String link = m.group(1).trim();
       System.out.println(link);
    }
```

有些链接的形式是：

```
    <a href='http://www.lietu.com'>猎兔</a>
```

为了更好地匹配单引号，可以把模式修改成：

```
    "<a\\s+href\\s*=\\s*[\"|']?(.*?)[\"|\'>]"
```

匹配“2009-12-6”这样的日期可以使用如下的正则表达式：

```
    Pattern p = Pattern.compile("\\d{2,4}-\\d{1,2}-\\d{1,2}");
    Matcher m = p.matcher(inputStr);
    if(m.find()){
       String strDate = m.group();
    }
```

其他的一些匹配日期的正则表达式有：“\\d{2,4}/\\d{1,2}/\\d{1,2}”和“\\d{2,4}年\\d{1,2}月\\d{1,2}日”，以及“\\d{2,4}\\.\\d{1,2}\\.\\d{2,4}”。

2.4　中文词语切分与词性标注

英语、法语和德语等西方语言通常采用空格或标点符号将词隔开，具有天然的分隔符，所以词的获取简单，但是为了深入地理解语义，仍然需要标注出每个词的词性。中文、日文和韩文等东方语言，虽然句子之间有分隔符，但词与词之间没有分隔符或者分隔符不够多，所以需要靠程序切分出词。

中文分词是中文自然语言理解的基础，这里首先介绍中文分词的接口与使用方法，然后介绍最长匹配中文分词和 *N* 元中文分词的实现。

2.4.1 使用中文分词

如果不需要切分出词性，则可以用如下简单的接口返回词：

```
String text = "科技进展";
Segmenter seg = new Segmenter(text);         //切分文本
String word;                                 //保存词
while ((word = seg.nextWord()) != null){     //返回一个词
   System.out.println(word);                 //输出切分出的词
};
```

如果需要标注出文本词性，则可以选用输出标注词性的分词接口。为了方便指明词的词性，词性标注程序往往给每个词性编码。例如，根据英文缩写，把“形容词”编码成 a，名词编码成 n，动词编码成 v……表 2-1 是完整的词性编码表。

表 2-1 词性编码表

代 码	名 称	举 例
a	形容词	最/d 大/a 的/u
ad	副形词	一定/d 能够/v 顺利/ad 实现/v 。/w
ag	形语素	喜/v 煞/ag 人/n
an	名形词	人民/n 的/u 根本/a 利益/n 和/c 国家/n 的/u 安稳/an 。/w
b	区别词	副/b 书记/n 王/nr 思齐/nr
c	连词	全军/n 和/c 武警/n 先进/a 典型/n 代表/n
d	副词	两侧/f 台柱/n 上/f 分别/d 雄踞/v 着/u
dg	副语素	用/v 不/d 甚/dg 流利/a 的/u 中文/nz 主持/v 节目/n 。/w
e	叹词	嗬/e ！/w
f	方位词	从/p 一/m 大/a 堆/q 档案/n 中/f 发现/v 了/u
g	语素	例如 dg 或 ag
h	前接成分	目前/t 各种/r 非/h 合作制/n 的/u 农产品/n
i	成语	提高/v 农民/n 讨价还价/i 的/u 能力/n 。/w
j	简称略语	民主/ad 选举/v 村委会/j 的/u 工作/vn
k	后接成分	权责/n 明确/a 的/u 逐级/d 授权/v 制/k
l	习用语	是/v 建立/v 社会主义/n 市场经济/n 体制/n 的/u 重要/a 组成部分/l 。/w
m	数词	科学技术/n 是/v 第一/m 生产力/n
n	名词	希望/v 双方/n 在/p 市政/n 规划/vn
ng	名语素	就此/d 分析/v 时/ng 认为/v
nr	人名	建设部/nt 部长/n 侯/nr 捷/nr
ns	地名	北京/ns 经济/n 运行/vn 态势/n 喜人/a

续表

代　码	名　称	举　例
nt	机构团体	[冶金/n　工业部/n　洛阳/ns　耐火材料/l　研究院/n]nt
nx	字母专名	ATM/nx　交换机/n
nz	其他专名	德士古/nz　公司/n
o	拟声词	汩汩/o　地/u　流/v　出来/v
p	介词	往/p　基层/n　跑/v　。/w
q	量词	不止/v　一/m　次/q　地/u　听到/v　，/w
r	代词	有些/r　部门/n
s	处所词	移居/v　海外/s　。/w
t	时间词	当前/t　经济/n　社会/n　情况/n
tg	时语素	秋/tg　冬/tg　连/d　旱/a
u	助词	工作/vn　的/u　政策/n
ud	结构助词	有/v　心/n　栽/v　得/ud　梧桐树/n
ug	时态助词	你/r　想/v　过/ug　没有/v
uj	结构助词的	迈向/v　充满/v　希望/n　的/uj　新/a　世纪/n
ul	时态助词了	完成/v　了/ ul
uv	结构助词地	满怀信心/l　地/uv　开创/v　新/a　的/u　业绩/n
uz	时态助词着	眼看/v　着/uz
v	动词	举行/v　老/a　干部/n　迎春/vn　团拜会/n
vd	副动词	强调/vd　指出/v
vg	动语素	做好/v　尊/vg　干/j　爱/v　兵/n　工作/vn
vn	名动词	股份制/n　这种/r　企业/n　组织/vn　形式/n　，/w
w	标点符号	生产/v　的/u　5G/nx　、/w　8G/nx　型/k　燃气/n　热水器/n
x	非语素字	生产/v　的/u　5G/nx　、/w　8G/nx　型/k　燃气/n　热水器/n
y	语气词	已经/d　30/m　多/m　年/q　了/y　。/w
z	状态词	势头/n　依然/z　强劲/a　；/w

使用词性编码输出给句子标注词性的结果，例如：“不/d 忘/v 群众/n 疾苦/n 温暖/v 送/v 进/v 万/m 家/q”。

调用输出词性标注的接口：

```
String text = "地球是一颗美丽的蓝色星球";

WordToken[] tokens = SentProcessor.tag(text);

System.out.println("输出标注结果：");
for (WordToken w : tokens) {
```

```
        System.out.println(w.term()+"|"+w.type());  //输出切分出来的词和词性
    }
```

2.4.2　正向最大长度匹配法

假如要切分“印度尼西亚地震”这个词组，希望切分出“印度尼西亚”，而不希望切分出“印度”这个词。正向找最长词是正向最大长度匹配的思想。倾向于写更短的词，除非必要，才用长词表述，所以倾向切分出长词。

正向最大长度匹配的分词方法实现起来很简单。每次从词典找和待匹配串前缀最长匹配的词，如果找到匹配词，则把这个词作为切分词，待匹配串减去该词，如果词典中没有词匹配上，则按单字切分。例如，检索树结构的词典中包括如下 8 个词语：

大　大学　大学生　活动　生活　中　中心　心

输入：“大学生活动中心”，首先匹配出开头的最长词“大学生”，然后匹配出“活动”，最后匹配出“中心”。切分过程如图 2-1 所示。



图 2-1　正向最大长度匹配切分过程

最后分词结果为：“大学生/活动/中心”。

在分词类 Segmenter 的构造方法中输入要处理的文本。然后通过 nextWord 方法遍历单词，其中，text 变量记录切分文本；offset 变量记录已经切分到哪里。分词类基本实现如下：

```
public class Segmenter {
   String text = null; //切分文本
   int offset; //已经处理到的位置

   public Segmenter(String text) {
      this.text = text; //更新待切分的文本
      offset = 0; //重置已经处理到的位置
   }

   public String nextWord() { //得到下一个词，如果没有，则返回 null
      //返回最长匹配词，如果没有匹配上，则按单字切分
   }
}
```

使用 Apache Commons Configuration 读入词表相关的配置信息，然后根据配置信息加载词表。build.gradle 文件增加依赖项：

```
dependencies {
    compile  group:  'org.apache.commons',  name:  'commons-configuration2',
version: '2.4'
    compile group: 'commons-beanutils', name: 'commons-beanutils', version:
'1.9.3'
}
```

将配置信息写入配置文件：

```
Configurations configs = new Configurations();
Configuration config = configs.properties(new File("config.properties"));
String dicDir = config.getString("dicDir");  //从配置文件读取词典路径
System.out.println(dicDir);
```

2.4.3　未登录串识别

切分结果中，英文和数字要连在一起，不管这些英文串或者数字串是否在词典中。例如“Twitter 正式发布音乐服务 Twitter#Music”这句话，即使词典中没有“Twitter”这个词，切分出来的结果也应该把 Twitter 合并在一起。另外，对于像[ATM 机]这样的英文和汉字混合的词也要合并在一起。

吃苹果时，比发现苹果中有一条虫更糟糕的是，发现里面只有半条虫。如果“007”在词表中，则会把“0078999”这样的数字串切分成多段。为了把一些连续的数字和英文切分到一起，需要区分全数字组成的词和全英文组成的词。如果匹配上了全数字组成

的词，则继续往后看还有没有更多的数字。如果匹配上了全英文组成的词，则继续往后看还有没有更多的字母。

匹配数字的有限状态自动机：

```
Automaton num = BasicAutomata.makeCharRange('0', '9').repeat(1);
num.determinize(); //转换成确定自动机
num.minimize(); //最小化
```

匹配英文单词的有限状态自动机：

```
Automaton lowerCase = BasicAutomata.makeCharRange('a', 'z');
Automaton upperCase = BasicAutomata.makeCharRange('A', 'Z');
Automaton c = BasicOperations.union(lowerCase, upperCase);
Automaton english = c.repeat(1);
english.determinize();
english.minimize();
```

匹配日期的有限状态自动机：

```
Automaton a = BasicAutomata.makeCharRange('0', '9');
Automaton b = a.repeat(2,4);
Automaton yearUnit = BasicAutomata.makeChar('年');
Automaton yearNum = BasicOperations.concatenate(b, yearUnit);
Automaton monUnit = BasicAutomata.makeChar('月');

Automaton twoNum = a.repeat(1,2);
Automaton monNum = BasicOperations.concatenate(twoNum, monUnit);
Automaton yearWithMon = BasicOperations.concatenate(yearNum,monNum);

Automaton dayUnit = BasicAutomata.makeChar('日');
Automaton dayNum = BasicOperations.concatenate(twoNum, dayUnit);
Automaton yearMonDay = BasicOperations.concatenate(yearWithMon,
      dayNum.optional());

Automaton finalDate = BasicOperations.union(yearNum,yearMonDay);
finalDate.determinize();
```

可以根据 Automaton 实例得到有限状态转换，例如得到匹配时间的有限状态转换：

```
public static FST createDate() throws Exception {
   Automaton dateAutomaton = AutomatonFactory.getCnDate();
   FST fstDate = new FST(dateAutomaton, "t"); //时间类型
   return fstDate;
}
```

如下方法返回同时匹配日期和数字的有限状态转换：

```
public static FST createAll()throws Exception  {
   FST dateFST = createDate(); //日期
   FST simpleFST = createSimple(); //数值
   FSTUnion union = new FSTUnion(dateFST, simpleFST);
   return union.union();
}
```

用于原子切分的 SplitPoints 类：

```
public class SplitPoints {
   public BitSet endPoints;              //可结束点
   public BitSet startPoints;            //可开始点
   public HashSet<POSType>[] atomPOS;

   public SplitPoints(int senLen) {
      endPoints = new BitSet(senLen);    //存储所有可能的切分点
      startPoints = new BitSet(senLen);  //存储所有可能的切分点
      atomPOS = new HashSet[senLen];     //存储可能的词性
   }

   @Override
   public String toString() {
      return "SplitPoints [endPoints=" + endPoints + "\r\n startPoints="
            + startPoints + "]";
   }
}
```

根据句子返回切分词图：

```
public AdjList getLattice(String sentence) {
   int atomCount = sentence.length();

   //原子切分
   SplitPoints splitPoints = fstSeg.splitPoints(sentence);

   AdjList g = new AdjList(atomCount + 1); //初始化在 Dictionary 中词组成的图
   sucNode = new CnToken[g.verticesNum];
   prob = new double[g.verticesNum];        //节点概率

   int start = 0;
   int currentEnd = splitPoints.endPoints.nextSetBit(0);

   while (start >= 0) {
      TernarySearchTrie.PrefixRet prefix = new TernarySearchTrie.PrefixRet();
      boolean matchRet = dic.matchAll(sentence, start, prefix, splitPoints.
endPoints);

      if (matchRet) {//匹配上
         for (WordEntry word : prefix.values) {
            int end = start + word.word.length();//词的结束位置
            double logProb = Math.log(word.freq) - Math.log(dic.n);
            CnToken tokenInf =
                  new CnToken(start, end, logProb, word.word, word.types);
            g.addEdge(tokenInf);
         }

         start = splitPoints.startPoints.nextSetBit(start + 1);
         currentEnd = splitPoints.endPoints.nextSetBit(currentEnd + 1);
```

```
            } else {
              double logProb = Math.log(1) - Math.log(dic.n);

              HashSet<POSType> types = null;

              if(splitPoints.atomPOS[start]==null ) {
                  types = new HashSet<POSType>();
                  types.add(new POSType("n", 1)); //默认为名词
              }
              else {
                  types = splitPoints.atomPOS[start];
              }

              g.addEdge(
      new CnToken(start, currentEnd, logProb, sentence.substring(start, currentEnd),
types));
              start = splitPoints.startPoints.nextSetBit(start + 1);
              currentEnd = splitPoints.endPoints.nextSetBit(currentEnd + 1);
            }
        }
        return g;
    }
```

2.4.4 基本的 *N* 元模型

两个词可以组合成一个词的情况叫作组合歧义。例如：“上海/银行”和“上海银行”。最大长度匹配算法无法正确切分组合歧义。例如，会把“请在一米线外等候”错误地切分成“一/米线”而不是“一/米/线”。

对于输入字符串 C“有意见分歧”，有下面两种切分可能：

S_1: 有/ 意见/ 分歧/

S_2: 有意/ 见/ 分歧/

这两种切分方法分别叫作 S_1 和 S_2。如何评价这两个切分方案？哪个切分方案更有可能在语料库中出现就选择哪个切分方案。

计算条件概率 $P(S_1|C)$和 $P(S_2|C)$，然后根据 $P(S_1|C)$和 $P(S_2|C)$的值来决定选择 S_1 还是 S_2。

因为联合概率 $P(C,S)= P(S|C)P(C)= P(C|S)P(S)$，所以有

$$P(S \mid C) = \frac{P(C \mid S) \times P(S)}{P(C)}$$。

这也叫作贝叶斯公式。$P(C)$是字串在语料库中出现的概率。比如说语料库中有 1 万个句子，其中有一句是“有意见分歧”那么 $P(C)$=P(“有意见分歧”)=万分之一。

在贝叶斯公式中，$P(C)$只是一个用来归一化的固定值，所以实际分词时并不需要计算。

$P(C|S)$是由从词串 S 恢复到汉字串 C 的概率，该值为 1，即 $P(C|S_1)=P(C|S_2)=1$。因此，比较 $P(S_1|C)$和 $P(S_2|C)$的大小变成比较 $P(S_1)$和 $P(S_2)$的大小，即

$$\frac{P(S_1 \mid C)}{P(S_2 \mid C)} = \frac{P(S_1)}{P(S_2)}$$

因为 $P(S_1)$=P(有,意见,分歧)> $P(S_2)$=P(有意,见,分歧)，所以选择切分方案 S_1 而不是 S_2。

在具体的分词过程中，输入是一个字符串 $C=C_1,C_2,\cdots,C_n$，输出是一个词串 $S=w_1,w_2,\cdots,w_m$，其中 $m \leqslant n$。对于一个特定的字符串 C，会有多个切分方案 S 与之对应，分词的任务就是在这些 S 中找出一个切分方案 S，使得 $P(S|C)$的值最大。$P(S|C)$就是由字符串 C 产生切分 S 的概率。最可能的切分方案为

$$\begin{aligned}\text{BestSeg}(C) &= \arg\max_{S\in G} P(S \mid C) = \arg\max_{S\in G} \frac{P(C \mid S)P(S)}{P(C)} \\ &= \arg\max_{S\in G} P(S) = \arg\max_{w_1,w_2,\cdots,w_m\in G} P(w_1,w_2,\cdots,w_m)\end{aligned}$$

也就是对输入字符串切分出最有可能的词序列。

这里的 G 表示切分词图。待切分字符串 C 中的某个子串构成一个词 w，把这个词看成是从开始位置 i 到结束位置 j 的一条有向边。把 C 中的每个位置看成点，词看成边，可以得到一个有向图，这个图就是切分词图 G。

概率语言模型分词的任务是：在全切分所得的所有结果中求某个切分方案 S，使得 $P(S)$为最大。那么，如何来表示 $P(S)$呢？为了简化计算，假设每个词之间的概率是上下文无关的，则

$$P(S) = P(w_1,w_2,\cdots,w_m) \approx P(w_1)P(w_2)\cdots P(w_m)$$

式中，$P(w)$就是词 w 出现在语料库中的概率。例如：

$$P(S_1) = P\,(\text{有,意见,分歧}) \approx P\,(\text{有})\,P(\text{意见})\,P(\text{分歧})$$

对于不同的 S，m 的值是不一样的。一般来说，m 越大，$P(S)$会越小。也就是说，分出的词越多，概率越小。这符合实际的观察，如最大长度匹配切分往往会使得 m 较小。

词表中的词往往很多，分摊到一个词的概率可能很小，所以 $P(S)$一般是通过很多小数值的连乘积算出来的。如果一个数太小，可能会向下溢出变成零。例如 0.00000000000000000000000000001，double 类型表示不出如此小的数。因为函数 $y=\log x$，当 x 增大时，y 也会增大，所以是单调递增函数。取对数后，表示一个小于 1 的正数的精确度加大了。

$P(S) \approx P(w_1)P(w_2)\cdots P(w_m) \propto \log P(w_1) + \log P(w_2) + \cdots + \log P(w_m)$，这里的$\propto$是正比符号。因为词的概率小于 1，所以取对数后是负数。最后算 $\log P(w)$。

计算任意一个词出现的概率如下：

$$P(w_i)=\frac{w_i\text{在语料库中的出现次数}n}{\text{语料库中的总词数}N}$$

因此 $\log P(w_i)=\log \mathrm{freq}_w-\log N$

如果词概率的对数值事前已经算出来了，则结果直接用加法就可以得到 $\log P(S)$，而加法比乘法的运算速度更快。

这个计算 $P(S)$的公式也叫作基于一元概率语言模型的计算公式。这种分词方法简称一元分词。它综合考虑了切分出的词数和词频。一般来说，词数少、词频高的切分方案概率更高。考虑一种特殊的情况：若所有词的出现概率相同，则一元分词退化成最少词切分方法。

句子切分的准确性在很大程度上取决于词语的上下文。比如，“上海银行间的拆借利率上升”，因“上海银行”后接词为“间”，这决定了“上海银行”应该切分为两个词“上海”和“银行”，而不是一个专有名词“上海银行”。

在一元分词中假设前后两个词的出现概率是相互独立的，但在实际中这不太可能。语言学家认为，一个词语的含义取决于它周围的词语。也就是说，某些词语会以很大概率经常出现在一起。比如，甜品店附近经常有咖啡馆，所以这两个词是正相关，但是很少会有人把“甜品店”和“沙县小吃”相提并论。[羡慕][嫉妒][恨]这三个词有时候会连续出现。切分出来的词序列越通顺，越有可能是正确的切分方案。N 元模型使用 n 个单词组成的序列来衡量切分方案的合理性。比如，估计单词 w_1 后出现 w_2 的概率，根据条件概率的定义

$$P(w_2\mid w_1)=\frac{P(w_1,w_2)}{P(w_1)}$$

可以得到 $P(w_1,w_2)=P(w_1)P(w_2|w_1)$

同理 $P(w1,w_2,w_3)=P(w_1,w_2)P(w_3|w_1,w_2)$

所以 $P(w_1,w_2,w_3)=P(w_1)P(w_2|w_1)P(w_3|w_1,w_2)$

更加一般的形式为

$$P(S)=P(w_1,w_2,\cdots,w_n)=P(w_1)P(w_2|w_1)P(w_3|w_1,w_2)\cdots P(w_n|w_1w_2\cdots w_{n-1})$$

这叫作概率的链规则。其中，$P(w_2|w_1)$表示 w_1 之后出现 w_2 的概率。如果词 w_1 和 w_2 独立出现，则 $P(w_2|w_1)$等价于 $P(w_2)$。

这样需要考虑在 n−1 个单词序列后出现的单词 w 的概率。直接使用这个公式计算 $P(S)$存在两个致命的缺陷：一是参数空间过大，不可能实用化；另外，是数据稀疏严重。例如，词汇量(V)=20000 时，可能的二元语法（bigram）组合数量有 400000000 个，可能的三元语法（trigram）组合数量有 8000000000000 个，可能的四元语法（4-gram）组合数量有 1.6×10^{17} 个。

为了解决这个问题，我们引入了马尔可夫假设：一个词的出现仅仅依赖于它前面出现的有限的一个或者几个词。

如果简化成一个词的出现仅依赖于它前面出现的一个词，那么就称为二元语法模型，即

$$P(S) = P(w_1,w_2,\cdots,w_n)= P(w_1)\, P(w_2|w_1)\, P(w_3|w_1,w_2)\cdots P(w_n|w_1w_2\ldots w_{n-1})$$

$$\approx P(w_1)\, P(w_2|w_1)P(w_3|w_2)\cdots P(w_n|w_{n-1})$$

例如，$P(S_1)$= P(有)P(意见|有)P(分歧|意见)，如果简化成一个词的出现仅依赖于它前面出现的两个词，就称之为三元语法模型。如果一个词的出现不依赖于它前面出现的词，称为一元语法（Unigram）模型，也就是已经介绍过的概率语言模型的分词方法。

如果切分方案 S 是由 n 个词组成的序列，那么 $P(w_1)P(w_2|w_1)P(w_3|w_2)\cdots P(w_n|w_{n-1})$也是 n 项连乘积。语言模型无论采用一元语法、二元语法还是三元语法都是 n 项连乘积。只不过二元语法以上模型是条件概率的连乘积。例如：对于切分“产品和服务”来说，二元语法模型计算为 P(产品)P(和|产品)P(服务|和)，三元语法模型计算为 P(产品)P(和|产品)P(服务|产品,和)。

因为 $P(w_i|w_{i-1})$= freq(w_{i-1},w_i)/freq(w_{i-1})，所以二元分词不仅用到二元词典，还需要用到一元词典。

概率语言模型中文分词切分过程说明如下。

（1）把输入字符串切分成句子：对一段文本进行切分，依次从这段文本中切分出一个个句子，然后对句子逐个进行分词。

（2）原子切分：对于一个句子的切分，首先是通过原子切分，将整个句子切分成一个个的原子单元（即不可再切分的形式，例如 Java 这样的英文单词可以被看成不可再切分的）。

（3）生成 n 元切分词图：根据基本词库对句子进行全切分，并且生成一个以邻接链表表示的基本词图。再根据基本词图得到 n 元词图。

（4）计算最佳切分路径：在这个词图的基础上，运用动态规划算法找出切分最佳路径。

（5）按 Lucene 和 Elasticsearch 定义的 API 输出结果。

二元切分词图简称二元词图，n 元切分词图简称 n 元词图。考虑如何得到二元词图：一个词的开始位置和结束位置组成的节点组合是二元词图中的点；前后两个词的转移概率作为边的权重。

例如，“有意见分歧”这句话中节点的组合有：{0,1}、{0,2}、{1,2}、{1,3}、{2,3}、{3,4}、{3,5}。得到的二元切分词图如图 2-2 所示。

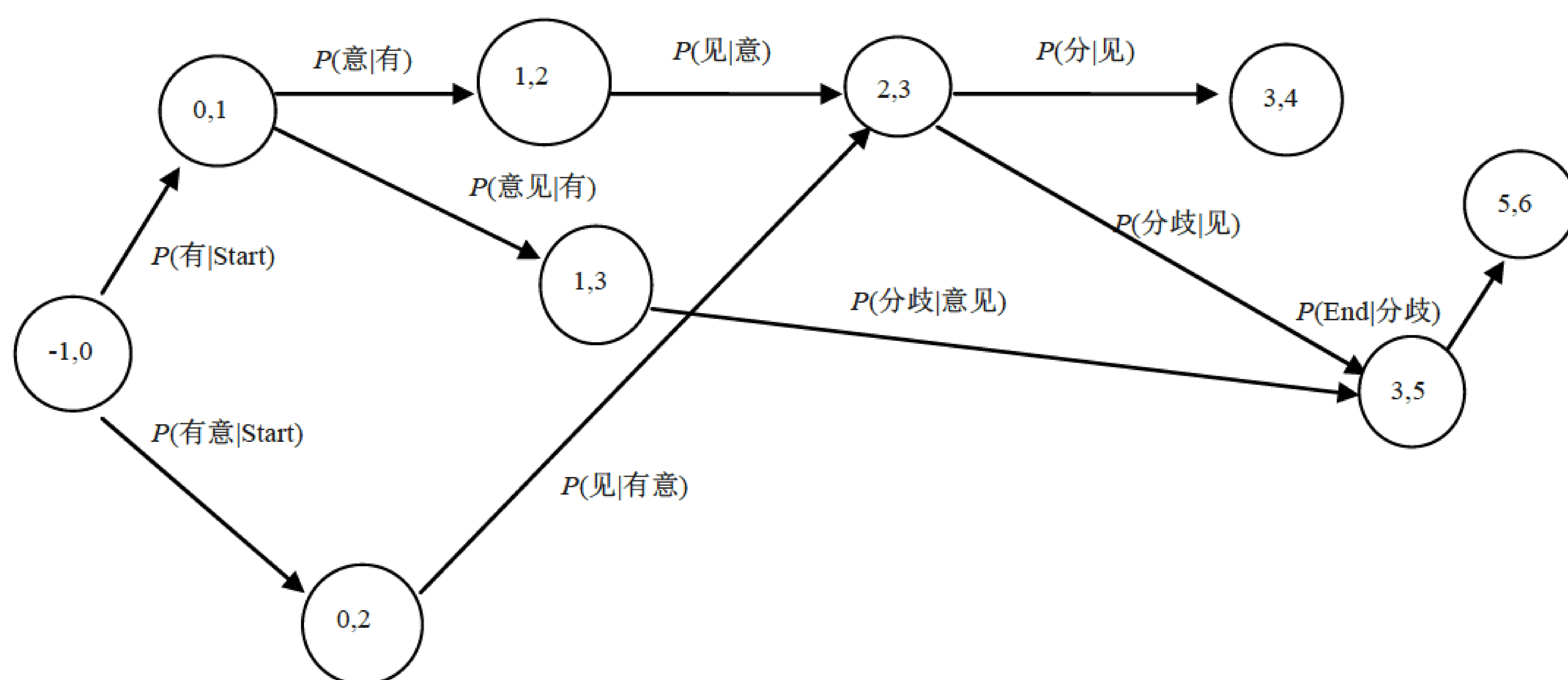


图 2-2　二元分词词图

Sqlite 数据库中存储的二元连接表的创建语句如下：

```
CREATE TABLE "BIGRAM" ("PREV" string NOT NULL DEFAULT (null) , --前一个词
"NEXT" string DEFAULT (null)                                    --后一个词
 ,"FRQ" int                                                     --二元频次
)
```

用于测试词条件概率的类：

```
public class DBBigramProb {
   Connection con;

   static double lambda1 = 0.3;       //平滑参数
   static double lambda2 = 0.7;

   public DBBigramProb() throws Exception {
      con = getSqliteConnection();
   }

   public double wordConditionalProbability(String prev,String next) {
   //条件概率
      QueryRunner runner = new QueryRunner();
      String sql = "select sum(frq) from UNIGRAM_WORD";
      Integer totalFreq = runner
             .query(con, sql, new ScalarHandler<Integer>());

      int dic_totalFreq = totalFreq;  //总频次

      int bigramFreq = getBigramFreq(prev, next);//从二元词典找二元频次

      int t1_freq = getFreq(next);
      int t2_freq = getFreq(prev);
```

```
        double wordProb = lambda1 * t1_freq / dic_totalFreq + lambda2
                * (bigramFreq / t2_freq);
        return wordProb;
    }

    private int getFreq(String t1) throws SQLException {
        QueryRunner runner = new QueryRunner();

        String sql = "select frq from UNIGRAM WORD where WORD =?";

        Integer freq = runner.query(con, sql, new ScalarHandler<Integer>(), t1);

        if (freq == null)
            return 0;

        return freq;
    }

    private int getBigramFreq(String t2, String t1)
            throws SQLException {  //二元连接频次
        QueryRunner runner = new QueryRunner();

        String sql = "select frq from BIGRAM where PREV =? and NEXT = ?";

        Integer freq = runner.query(con, sql, new ScalarHandler<Integer>(), t2,
                t1);

        if (freq == null)
            return 0;

        return freq;
    }

    public static Connection getSqliteConnection() throws Exception {
                                    //得到数据库连接
        try {
            Class.forName("org.sqlite.JDBC");
            String absolute path to sqlite db = "./dic/bigram.sqlite";
                                    //二元词库
            return DriverManager.getConnection("jdbc:sqlite:"
                    + absolute_path_to_sqlite_db);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

使用这个测试类：

```
String prev = "有";
```

```
String next = "意见";

DBBigramProb bigramProb = new DBBigramProb();
double wordCondProb = bigramProb.wordConditionalProbability(prev, next);

double logProb = Math.log(wordCondProb);
System.out.println(logProb);
```

组合节点类定义如下：

```
public class Node {
   public int start;                                  //开始节点
   public int end;                                    //结束节点

   public double logProb;                             //节点本身的概率
   public int frq;                                    //组合频次
   public double nodeProb;                            //节点累积概率
   public Node bestPrev;                              //最佳前驱节点

   public Node(int s, int e, int fq, double p) {      //构造方法
      start = s;
      end = e;
      frq = fq;
      logProb = p;
   }

   @Override
   public int hashCode() {
      return start ^ end;
   }

   @Override
   public boolean equals(Object o) {
      if (!(o instanceof Node)) //判断传入对象的类型
        return false;
      Node that = (Node) o;

      return (this.start == that.start && this.end == that.end);
   }

   public String toString() {
      return start + ":" + end+" frq "+frq+" logProb "+logProb;
   }
}
```

LatticeFactory 类生成词图：

```
public class LatticeFactory {
   public static TernarySearchTrie dic = null;    //词典树
   static FSTSGraph fstSeg;                       //用于原子切分的有限状态转换
```

```
        static {
           try {
              fstSeg = new FSTSGraph();
           } catch (Exception e) {
              e.printStackTrace();
           }

           dic = (new DicFileFactory()).create();      //创建词典树
           //如果要根据数据库中的表创建词典树，则使用如下语句
           //(new DicDBFactory()).create();
        }

        public static AdjList getLattice(String text) throws Exception {
           int sLen = text.length();                   //字符串长度
           AdjList g = new AdjList(sLen + 2);          //存储所有被切分的可能的词

           //原子切分
           SegScheme schema = fstSeg.seg(text);

           //用来存放前驱词的集合
           ArrayList<WordEntry> prevWords = new ArrayList<WordEntry>();

           //从前往后求出每个节点的最佳前驱节点和它的节点概率
           int fromPoint = 0;
           while (fromPoint >= 0) {
              int start = schema.startPoints.nextSetBit(fromPoint);     //开始点
              int end = schema.endPoints.nextSetBit(fromPoint + 1);     //结束点

              fromPoint = schema.startPoints.nextSetBit(start + 1);

              //从词典中查找前驱词的集合
              boolean match = dic.matchAll(text, start, prevWords,
                      schema.endPoints);

              if (!match) {
                  //词典中找不到对应的词,则返回开始点和结束点之间的字符串
                  String word = text.substring(start, end);
                  prevWords.add(new WordEntry(word, 1));
                  Node newEdge = new Node(start, start + word.length(),1,
                          Math.log((double) 1 / dic.n));
                  g.addEdge(newEdge);       //词图增加边
              } else {
                  for (WordEntry w : prevWords) {  //遍历找到的每个词
                      Node newEdge = new Node(start, start + w.word.length(),w.
freq,
                              Math.log((double) w.freq / dic.n));
                      g.addEdge(newEdge);  //词图增加边
                  }
              }
```

```
        }

        return g;
    }
}
```

Segmenter.bestPrev()方法从后往前计算词图中每个节点的最佳前驱节点：

```
public static AdjList bestPrev(AdjList wordGraph) throws Exception {
    for (Node currentNode : wordGraph) { //从前往后遍历切分词图中的每个节点

        //得到当前节点的前驱节点集合
        NodeLinkedList prevNodes = wordGraph.prevNodes(currentNode);

        double nodeProb = minValue;         //候选词概率
        Node minNode = null;
        if (prevNodes == null) {
            currentNode.nodeProb = 0;
            continue;
        }
        for (Node prevNode : prevNodes) {
            double currentProb = transProb(prevNode, currentNode) //转移概率
                    + prevNode.nodeProb;     //前一个节点的节点概率

            if (currentProb > nodeProb) {
                nodeProb = currentProb;
                minNode = prevNode;
            }
        }
        currentNode.bestPrev = minNode;     //设置当前节点的最佳前驱节点
        currentNode.nodeProb = nodeProb;    //设置当前节点的节点概率
    }

    return wordGraph;
}
```

二元分词方法切分文本的代码如下：

```
public static List<String> split(String text) throws Exception {
    AdjList wordGraph = LatticeFactory.getLattice(text);   //得到词图
    bestPrev(wordGraph);                        //从后往前计算最佳前驱节点
    ArrayDeque<Node> seq = new ArrayDeque<Node>();     //切分出来的节点序列
    //从后向前找最佳前驱节点
    for (Node t = wordGraph.endNode.bestPrev; t.start > -1; t = t.bestPrev) {
        seq.addFirst(t);
    }
    //根据最佳前驱节点数组回溯求解词序列
    return bestPath(text, seq);
}
```

测试分词的代码如下：

```
String sentence = "中国成立了";
```

```
List<String> ret = Segmenter.split(sentence);
System.out.println("切分结果 ");
for (String word : ret) {
   System.out.print(word+" / ");
}
```

2.5　隐马尔可夫模型

可以使用隐马尔可夫模型（hidden markov model，HMM）实现词性标注。

2.5.1　数据基础

词典要能够识别每个词可能的词性。例如，可以根据词性编码在文本文件 n.txt 中存放名词，在文本文件 v.txt 中存放动词，在文本文件 a.txt 中存放形容词，等等。例如，v.txt 的内容如下：

```
欢迎
迎接
```

可以把这些按词性分放到不同文件的词表合并成一个大的词表文件，每行一个词和对应的一个词性。例如：

```
把:p
把:q
```

如果把词表放到数据库中，则设置词和词性两列。为了避免重复插入词，词和词性联合作为主键。

```
CREATE TABLE "AI_BASEWORDS" ("WORD" string NOT NULL , --词
 "PARTSPEECH" string,  --词性
 "FRQ" int, --词频
 "PINYIN" string) --拼音
```

从 MySQL 数据库读出词的代码如下：

```
TernarySearchTrie dic = new TernarySearchTrie();

Connection conn = getConnect();        //得到数据库连接

String sql = "SELECT WORD,PARTSPEECH,FRQ from AI BASEWORD";
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery(sql);

while (rs.next()) {
   String word = rs.getString(1);
```

```
        String pos = rs.getString(2);
        int frq = rs.getInt(3);

        if(frq<=0) {
           System.out.println("词频错误 "+word +" frq "+frq);
           frq = 1;
        }

        dic.addWord(word, pos,frq);   //增加词表到词典树
        dic.n +=frq;                   //总频次
     }

     conn.close();
```

2.5.2 维特比算法

解决标注歧义问题最简单的一种方法是从单词所有可能的词性中选出这个词最常用的词性作为这个词的词性，也就是一个概率最大的词性，比如“改革”大部分时候作为一个名词出现，那么可以机械地把这个词总是标注成名词，但是这样标注的准确率会比较低，因为只考虑了频率特征。

考虑词所在的上下文可以提高标注的准确率。一般，在动词后接名词的概率很大。例如，“推进/改革”中的“推进”是动词，所以后面的“改革”很有可能是名词。这样的特征叫作上下文特征。

隐马尔可夫模型和基于转换的学习方法是两种常用的词性标注方法。这两种方法都整合了频率和上下文两方面的特征来取得好的标注结果。具体来说，隐马尔可夫模型同时考虑到了词的生成概率和词性之间的转移概率。

很多生物也懂得同时利用两种特征信息。例如，箭鼻水蛇是一种生活在水中以吃鱼或虾为生的蛇。它是唯一一种长着触须的蛇类。箭鼻水蛇最前端的触须能够感触非常轻微的变动，这表明它可以感触鱼类移动时产生的细微水流变化。在光线明亮的环境中，箭鼻水蛇能够通过视觉捕食小鱼。因此，它能够同时利用触觉和视觉，即通过光线的变化和水流的变化信息来捕鱼。

词性标注的任务是：给定词序列 $W=w_1,w_2,\cdots,w_n$，寻找词性标注序列 $T=t_1,t_2,\cdots,t_n$，使得 $P(t_1,t_2,\cdots,t_n|w_1,w_2,\cdots,w_n)$这个条件概率最大。

例如，词序列是：[他] [会] [来]这句话。为了简化计算，假设只有词性：代词（r）、动词（v）、名词（n）和方位词（f）。这里：[他]只可能是代词，[会]可能是动词或者名词，而[来]可能是方位词或者动词。所以有 4 种可能的标注序列。需要比较：P(r,v,v|他,会,来)、P(r,n,v|他,会,来)、P(r,v,f|他,会,来)和 P(r,n,f|他,会,来)，发现 P(r,v,v|他,会,来)是这 4 个概率中最大的，所以选择词性标注序列[r,v,v]。

使用贝叶斯公式重新描述这个条件概率：

$$P(t_1,t_2,\cdots,t_n)P(w_1,w_2,\cdots,w_n|t_1,t_2,\cdots,t_n)/P(w_1,w_2,\cdots,w_n)$$

忽略掉分母 $P(w_1,w_2,\cdots,w_n)$。

$$P(t_1,t_2,\cdots,t_n)= P(t_1)P(t_2|t_1)P(t_3|t_1,t_2)\cdots P(t_n|t_1t_2\cdots t_{n-1})$$

做独立性假设，使用 n 元模型近似计算 $P(t_1,t_2,\cdots,t_n)$。例如使用二元语法模型，则有

$$P(t_1,t_2,\cdots,t_n)\approx\prod_{i=1}^{n}P(t_i\mid t_{i-1})$$

近似计算 $P(w_1,w_2,\cdots,w_n|t_1,t_2,\ \cdots,t_n)$：假设一个类别中的词独立于它的邻居，则有

$$P\left(w_1,w_2,\cdots,w_n\mid t_1,t_2,\cdots,t_n\right)\approx\prod_{i=1}^{n}P(w_i\mid t_i)$$

寻找最有可能的词性标注序列实际的计算公式为

$$P(t_1,t_2,\cdots,t_n)P(w_1,w_2,\cdots,w_n|t_1,t_2,\cdots,t_n)\approx\prod_{i=1}^{n}P(t_i\mid t_{i-1})P(w_i\mid t_i)$$

因为词是已知的，所以这里把词 w 称为显状态。因为词性是未知的，所以把词性 t 称为隐状态。条件概率 $P(t_i|t_{i-1})$称为隐状态之间的转移概率。条件概率 $P(w_i|t_i)$称为隐状态到显状态的发射概率，也称为隐状态生成显状态的概率。注意，不要把 $P(w_i|t_i)$算成 $P(t_i|w_i)$。

因为出现某个词性的词可能很多，所以对很多词来说，发射概率 $P(w_i|t_i)$往往很小。而词性往往只有几十种，所以转移概率 $P(t_i|t_{i-1})$往往比较大。就好像这世界有各种各样的动物，在所有的动物中，正好碰到啄木鸟的可能性比较小。

使用 byte 类型表示一个词性，定义词性的 POS 类实现如下：

```
public class POS {
   public final static byte start = 0;          //开始
   public final static byte end = 1;            //结束
   public final static byte a = 2;              //形容词
   public final static byte ad = 3;             //副形词
   public final static byte ag = 4;             //形语素
   public final static byte an = 5;             //名形词
   public final static byte b = 6;              //区别词
   public final static byte c = 7;              //连词
   public final static byte d = 8;              //副词
   public final static byte dg = 9;             //副语素
   public final static byte e = 10;             //叹词
   public final static byte f = 11;         //方位词
   public final static byte g = 12;         //语素
```

```
    public final static byte h = 13;        //前接成分
    public final static byte i = 14;        //成语
    public final static byte j = 15;        //简称略语
    public final static byte k = 16;        //后接成分
    public final static byte l = 17;        //习用语
    public final static byte m = 18;        //数词
    public final static byte n = 19;        //名词
    public final static byte ng = 20;       //名语素
    public final static byte nr = 21;       //人名
    public final static byte ns = 22;       //地名
    public final static byte nt = 23;       //机构团体
    public final static byte nx = 24;       //字母专名
    public final static byte nz = 25;       //其他专名
    public final static byte o = 26;        //拟声词
    public final static byte p = 27;        //介词
    public final static byte q = 28;        //量词
    public final static byte r = 29;        //代词
    public final static byte s = 30;        //处所词
    public final static byte t = 31;        //时间词
    public final static byte tg = 32;       //时语素
    public final static byte u = 33;        //助词
    public final static byte ud = 34;       //结构助词
    public final static byte ug = 35;       //时态助词
    public final static byte uj = 36;       //结构助词的
    public final static byte ul = 37;       //时态助词了
    public final static byte uv = 38;       //结构助词地
    public final static byte uz = 39;       //时态助词着
    public final static byte v = 40;        //动词
    public final static byte vd = 41;       //副动词
    public final static byte vg = 42;       //动语素
    public final static byte vn = 43;       //名动词
    public final static byte w = 44;        //标点符号
    public final static byte x = 45;        //非语素字
    public final static byte y = 46;        //语气词
    public final static byte z = 47;        //状态词
    public final static byte unknow = 48; //未知
}
```

存储词的转移频次的 POSTransFreq.txt 文件内容如下：

```
start:uj:1
start:v:5230
start:vd:4
start:vg:54
```

```
start:vn:440
start:w:5047
start:y:2
start:z:58
a:end:173
a:a:833
a:ad:8
a:ag:6
a:an:127
a:b:62
a:c:451
a:d:296
a:dg:2
a:f:84
a:i:13
a:j:80
a:k:4
a:l:125
a:m:896
```

Tagger 类中的属性如下：

```
//转移频次
private int[][] transFreq = new int[POS.names.length][POS.names.length];
//每个词性的频次
private int[] typeFreq = new int[POS.names.length];
private int totalFreq;                                //所有词的总频次
```

读取 POSTransFreq.txt 文件，得到类型频次和转移频次的代码如下：

```
String transFrq = "./dic/POSTransFreq.txt";
InputStream file = new FileInputStream(new File(transFrq));

BufferedReader read = new BufferedReader(new InputStreamReader(
      file, "GBK"));

String line = null;

while ((line = read.readLine()) != null) {
   StringTokenizer st = new StringTokenizer(line, ":");
   int pre = POS.values.get(st.nextToken());       //前面的词类
   int next = POS.values.get(st.nextToken());      //后面的词类
   int frq = Integer.parseInt(st.nextToken());     //词频
   transFreq[pre][next] = frq;                     //转移频次
   typeFreq[next] += frq;                          //类型频次
   totalFreq += frq;
   //如果有从开始类型到其他类型的转移频次，就把这样的转移频次计入开始频次的类型频次
   if (pre == 0) {
      typeFreq[0] += frq;
   }
}
read.close();
```

为了避免乘积项为零，平滑转移概率的公式为

$$P(t_i \mid t_j) = \lambda_1 \frac{\text{Freq}(t_j, t_i)}{\text{Freq}(t_j)} + \lambda_2 \frac{\text{Freq}(t_j)}{\text{Freq}(\text{total})}$$

这里，取 λ_1=0.9， λ_2=0.1。

根据平滑公式计算转移概率的 getTransProb()方法实现如下：

```
/**
 * 计算上一个到下一个词的转移概率
 * @param curState
 *            前一个词性
 * @param toTranState
 *            后一个词性
 * @return
 */
public double getTransProb(byte curState, byte toTranState) {
   return Math.log((0.9 * transFreq[curState][toTranState]
        / typeFreq[curState] + 0.1 * typeFreq[curState] / totalFreq));
}
```

2.6 英文文本切分与标注

这里首先介绍英文句子切分的方法，然后介绍英文词性标注。

2.6.1 句子切分

英文句子切分并不是一个简单的问题。标点符号“?”和“!”的含义比较单一。但是“.”有很多种不同的用法，并不一定是句子的结尾。例如：“Mr. Vinken is chairman of Elsevier N.V.,the Dutch publishing group.”需要排除掉一部分情况。如果“.”是某个短语中间的一部分，则它不是句子的结尾。这里的“Mr. Vinken”是一个人名短语。如果这个人名正好不在词典中，则可以根据上下文识别规则识别出这个短语。

```
    String text= "Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing group.";
    EnText enText = new EnText(text);
    for(Sentence sent:enText){
       System.out.println(sent); //因为输入的是一个句子,所以这里只会打印出一个句子
    }
```

Java 中的 BreakIterator 类已经包含了切分句子的功能。用它实现一个英文句子迭代器：

```
private final static class SentBreakIterator implements Iterator<Sentence> {
   String text;
   int start;
   int end;
   //根据英文标点符号切分
   static final BreakIterator boundary = BreakIterator
         .getSentenceInstance(Locale.ENGLISH);

   public SentBreakIterator(String t) {
      text = t;
      //设置要处理的文本
      boundary.setText(text);
      start = boundary.first(); //开始位置
      end = boundary.next();
   } //用于迭代的类

   @Override
   public boolean hasNext() {
      return (end != BreakIterator.DONE);
   }

   @Override
   public Sentence next() {
      String sent = text.substring(start, end);

      Sentence sentence = new Sentence(sent, start, end);
      start = end;
      end = boundary.next();
      return sentence;
   }
}
```

BreakIterator 分得不太准确。所以我们自己写一个句子切分器。输入当前切分点，找下一个切分点的代码如下：

```
public static int nextPoint(String text, int lastEOS) {
   int i = lastEOS;
   while (i < text.length()) {
      //跳过短语
      i = skipPhrase(text, i);

      //然后再找标点符号
      String toFind = eosDic.matchLong(text, i); //匹配标点符号词典
      if (toFind != null) {
        //判断是否有效的可切分点。例如，在括号中的标点符号不是有效的可切分点
        boolean isEndPoint = isSplitPoint(text, lastEOS, i);
        if (isEndPoint) {
            return i + toFind.length();
        }
```

```
            i = i + toFind.length();
        } else { //没找到
            i++;
        }
    }
    return text.length(); //返回最大长度
}
```

SentIterator 是一个用于迭代英文文本返回句子的内部类，实现代码如下：

```
private final static class SentIterator implements Iterator<Sentence> {
    String text;
    int lastEOS = 0;

    public SentIterator(String t) {
        text = t;
    }

    @Override
    public boolean hasNext() {
        return (lastEOS < text.length());
    }

    @Override
    public Sentence next() {
        int nextEOS = EnSentenceSpliter.nextPoint(text, lastEOS);
        String sent = text.substring(lastEOS, nextEOS);
        Sentence sentence = new Sentence(sent, lastEOS, nextEOS);
        lastEOS = nextEOS;
        return sentence;
    }
}
```

2.6.2 标注词性

一段英文：Cats never fail to fascinate human beings. They can be friendly and affectionate towards humans,but they lead mysterious lives of their own as well.

标注词性后的结果是：

```
    Cats(n.) never fail(v.) to(prep) fascinate(v.) human(n.) beings(n.).
They(pron.) can(aux.) be(v.) friendly(adj.) and(conj.) affectionate(adj.)
towards(prep) humans(n.) but(conj.) they(n.) lead(v.) mysterious(adj.) lives(n.)
of(prep.) their(n.) own(n.) as well(adv.).
```

这里用编码来表示词性。括号中的输出是词性编码。汉语中的量词是英语中没有的，例如件、个、艘。英语中也有一些独有的词性，例如冠词 a、an、the。英文词性编码表如表 2-2 所示。

表 2-2 英文词性编码表

代　　码	名　　称
n	名词
adj	形容词
adv	副词
art	冠词
pos	所有格
pron	代词
aux	情态助动词
conj	连接词
v	动词
num	数词
prep	介词
punct	标点符号
int	感叹词

词性标注的流程图如图 2-3 所示。

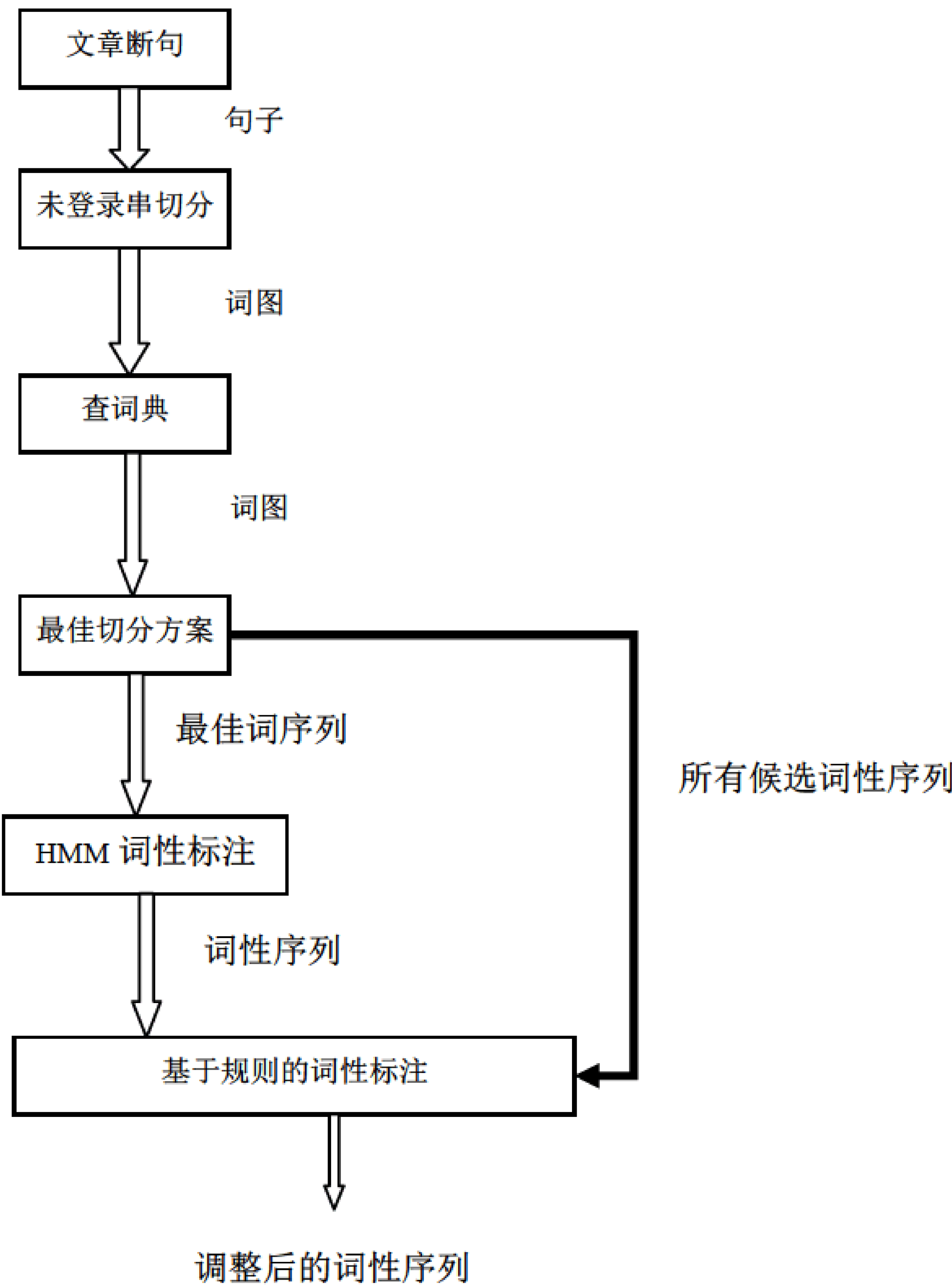


图 2-3 词性标注流程图

关于利用隐马尔可夫模型做词性标注在中文词性标注实现中已经介绍过了。英文词性标注语料库和中文词性标注语料库不同。

标注规则例如 I like it 对应的词性序列[r v r]。

```
key = new ArrayList<PartOfSpeech>();
key.add(PartOfSpeech.pron);  //I
key.add(PartOfSpeech.v);     //like
key.add(PartOfSpeech.pron);  //it
posTrie.addProduct(key);
```

实现代码：

```
public static ArrayList<WordToken> getWords(Sentence sent){
   ArrayList<WordTokenInf> words = Segmenter.seg(sent); //先分词
   WordType[] tags = g.tag(words); //然后标注词性

   //再把词性和词本身结合起来，返回完整的词性标注结果
   int i=0;
   ArrayList<WordToken> tokens = new ArrayList<WordToken>();

   for(WordTokenInf w:words){
     WordToken t = new WordToken(w.baseForm,w.termText,w.start,w.end,tags[i]);

      ++i;
      tokens.add(t);
   }

   return tokens;
}
```

2.7 命名实体识别

命名实体识别包括人名识别、地名识别和机构名识别等。因人名、地名和机构名的识别方法基本相同，故本书主要介绍人名识别。

2.7.1 人名识别

中文文本中的未登录人名包括中国人名和外国译名以及日本人名等。例如，“彭帅、郑洁 1∶2 不敌阿根廷选手杜尔科和意大利选手佩内塔”，其中包括中国人名{彭帅、郑洁}，还有外国人名“杜尔科”和“佩内塔”。

英文人名的全名是由 first name，middle name 和 last name 三部分组成的，其中 middle name 的主要目的只是为了防止重名，一般生活中会省略中间名。

对于没有能够根据词典与相邻字组成 2 个字以上词的字符，切分出来的结果称为切分碎片。例如，“素与杨宝森先生交好”，如果“杨宝森”这个词不在词典中，则切分出来的结果是：“素/与/杨/宝/森/先生/交好”。

人名往往在切分碎片中，但是也有特例：

- 人名内部相互成词。指姓与名、名与名之间本身就是一个已经被收录的词。例如，[王国]维、[高峰]、[汪洋]、张[朝阳]、冯[胜利]。
- 人名与其上下文组合成词。例如，“这里[有关]天培的壮烈”。

对识别人名有用的信息：

- 人名所在的上下文。例如：“**教授”，这里“教授”是人名的下文；“邀请**”，这里“邀请”是人名的上文。
- 人名本身的概率。例如：不依赖上下文，直观地来看，“刘宇”可能是个人名，而“史光”不太可能是个人名。若采用未登录词的概率作为这种可能性的衡量依据，则“刘宇”作为人名的概率是：“刘宇”作为人名出现的次数/人名出现的总次数。怎么算当前这个人名的出现概率？用姓的概率×名字的概率。

2.7.2 人名识别规则

人名识别有一些规则，例如“让<nr>和<nr>一起”。

分析中国人名所在的上下文，表明身份的词有：

- 出现在人名之前的词：工人、教师、影星、犯人。
- 出现在人名之后的词：先生、同志。
- 既可能出现在前面，也可能出现在后面的词：校长、经理、主任、医生。

地名或机构名往往出现在人名之前，例如：静海县大丘庄禹作敏。

“的”字结构往往出现在人名之前，例如：年过七旬的王贵芝。

有的动作词出现在人名之前，例如：批评，逮捕，选举。有的动作词出现在人名之后，例如：说，表示，吃，结婚。

未登录人名识别过程是：首先从输入串找所有可能的人名，然后再按照 N 元模型做分词，过滤候选人名。例如输入串原文是：程正泰的父亲是一位京剧票友，素与杨宝森先生交好。从中提取出候选人名{程正泰,杨宝,杨宝森}，把候选人名“杨宝”过滤掉。

识别候选人名的两层信息：底层是组成未登录姓名的单字特征和上下文词特征，特征串作为一个整体的搭配。

未登录中国人名相关的特征见表 2-3。

最容易想到的方法是：先找姓，然后找名。但有的未登录姓名只有名字，没有姓。

表 2-3　未登录中国人名相关的特征

编　　号	特　　征	举　　例
1	姓	张/nr　卫涛/nr
2	双名首字	张/nr　卫涛/nr
3	双名尾字	张/nr　卫涛/nr
4	单名	赵/nr　红/nr
11	上文	主席/n　李/nr　贤哲/nr
12	下文	李/nr　贤哲/nr　首先/d
13	同时做上文和下文	李/nr　贤哲/nr　和/d　陈/nr　涛/nr

把人名特征存放在 nr.txt 文本中。根据特征词表 nr.txt 对输入串全切分，形成人名特征词图。采用邻接链表（AdjList）存储切分结果。也就是说，用邻接链表表示人名特征词图。例如输入串“我爸是李刚”组成的特征词图如图 2-4 所示。

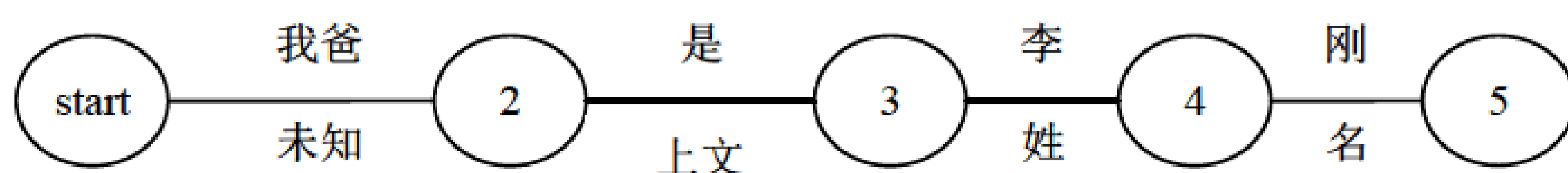


图 2-4　人名特征词图

根据特征序列识别出人名。例如：我爸叫李刚。这里[动词,姓,名,标点符号]组成了一个包含中国人名的特征序列。把[动词,姓,名,标点符号]称为一个识别规则。可以根据这个识别规则识别出“李刚”这个人名。这个规则的完整形式是：

动词　中国人名　标点符号 ⟹ 动词　姓　名　标点符号

例如有一条识别规则[姓氏,单名,下文]，用特征编号序列表示是[1,4,12]。

一个词可以同时是两种类型。例如“彭帅、郑洁 1∶2 不敌阿根廷选手杜尔科和意大利选手佩内塔”这句话，这里的[、]既是[彭帅]这个人名的下文，[、]也是[郑洁]这个人名的上文。

因为未登录人名往往在分词结果中出来的是切分碎片，所以最简单的方法是把分词结果再标注一次人名特征，如图 2-5 所示。其中的人名特征用编号表示。

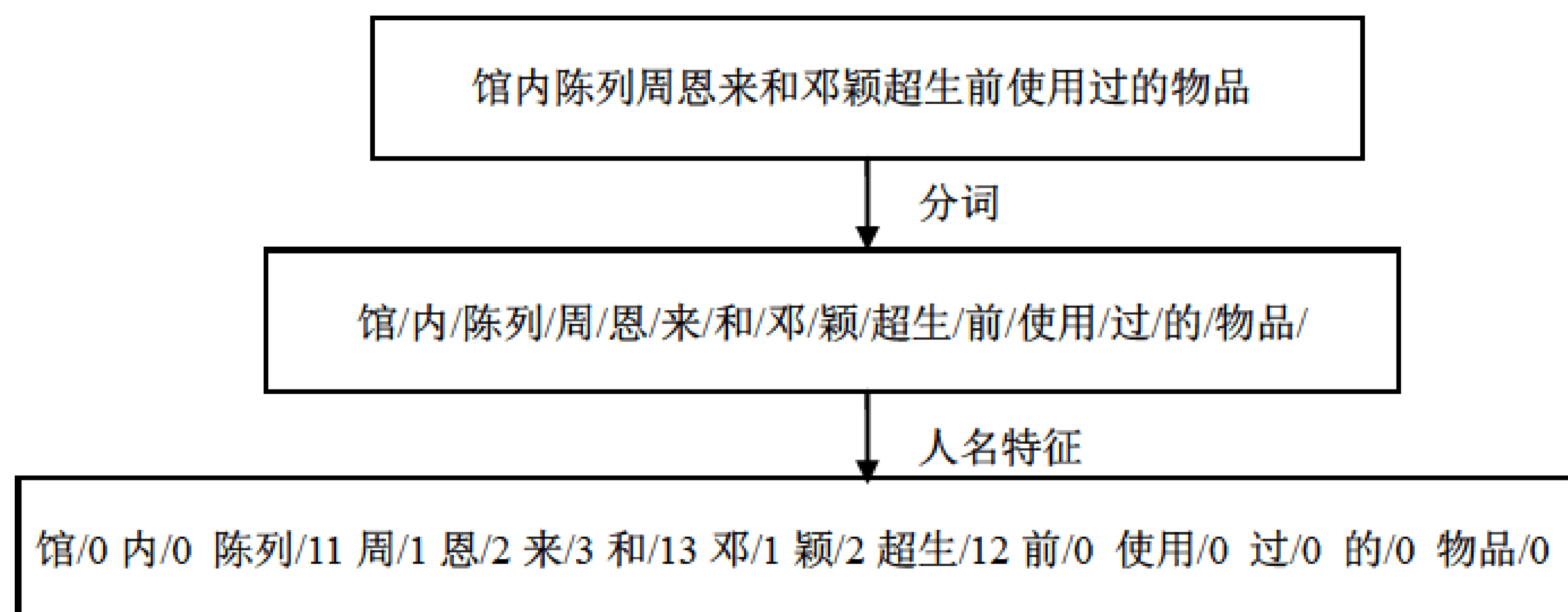


图 2-5　从分词序列中找人名特征

因为“超生”形成了一个词，所以用词序列只能识别出“邓颖”这样的人名，无法正确的识别出“邓颖超”，所以用人名特征专用的词图，如图 2-6 所示。也就是说，用存放在 nr.txt 中的人名特征词切分出人名特征词图。

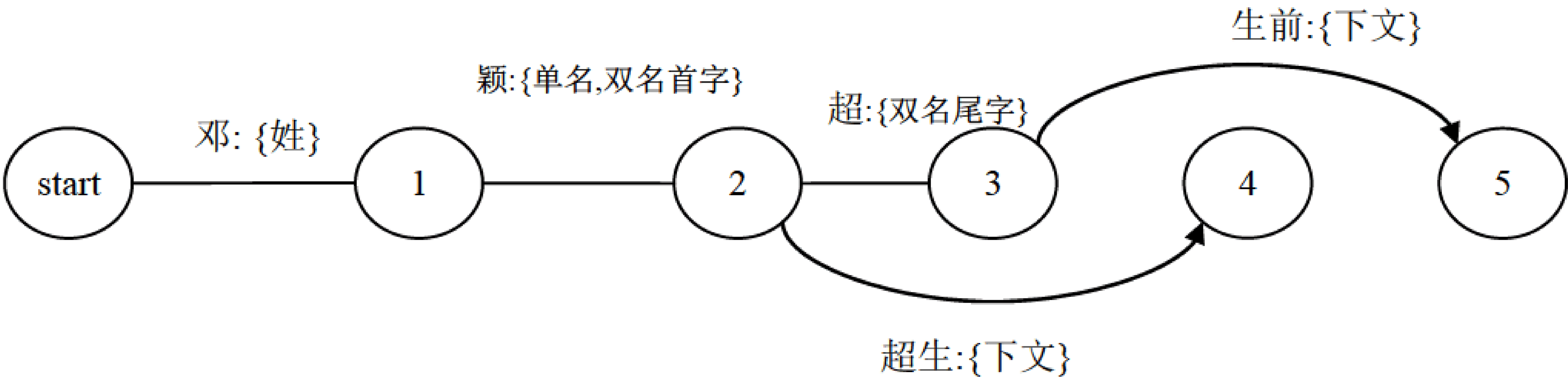


图 2-6　“邓颖超生前”特征词图

在图 2-6 所示的切分特征图里找到人名的识别规则序列[1,2,3,12]，找到后就能把“邓颖超”识别成一个人名。也就是从特征词图找到[邓,颖,超,生前]对应的特征序列[姓氏,双名首字,双名尾字,下文]。

首先有一些和识别未登录词相关的特征词表，然后输入串根据特征词表形成特征词图。最后根据未登录词识别规则从特征词图中找候选未登录词。

人名规则，例如[姓+名]，类似的规则还有很多，所以可用有限状态自动机求交集的方法同时找出所有可能的人名。存在一个句子中的人名特征词图，还有一个是由规则树组成的检索树（trie tree）。人名特征词图（也就是一个 AdjList 的实例）相当于一个 DFA，规则树组成的检索树相当于另外一个 DFA。找候选人名相当于对这两个 DFA 求交集。

在特征词序列上识别人名，不是在原始的字符上识别人名。特征词类型定义成为枚举类型：

```
public enum PersonType {
    preContext,         //上下文中的上文，例如，邀请**
    postContext,        //上下文中的下文，例如，**说
    surName,            //姓
    singleName,         //单名
    doubleName1,        //双名第一个字
    doubleName2         //双名第二个字
}
```

例如：[老生] [虞] [子] [期] [小生]。这里的[老生]是 preContext，而[小生]则是 postContext。

人名识别规则有很多，所以用标准检索树存储规则。例如，有规则[1,4,12]，[11,1,2,3]，[11,1,4,12]，如图 2-7 所示。

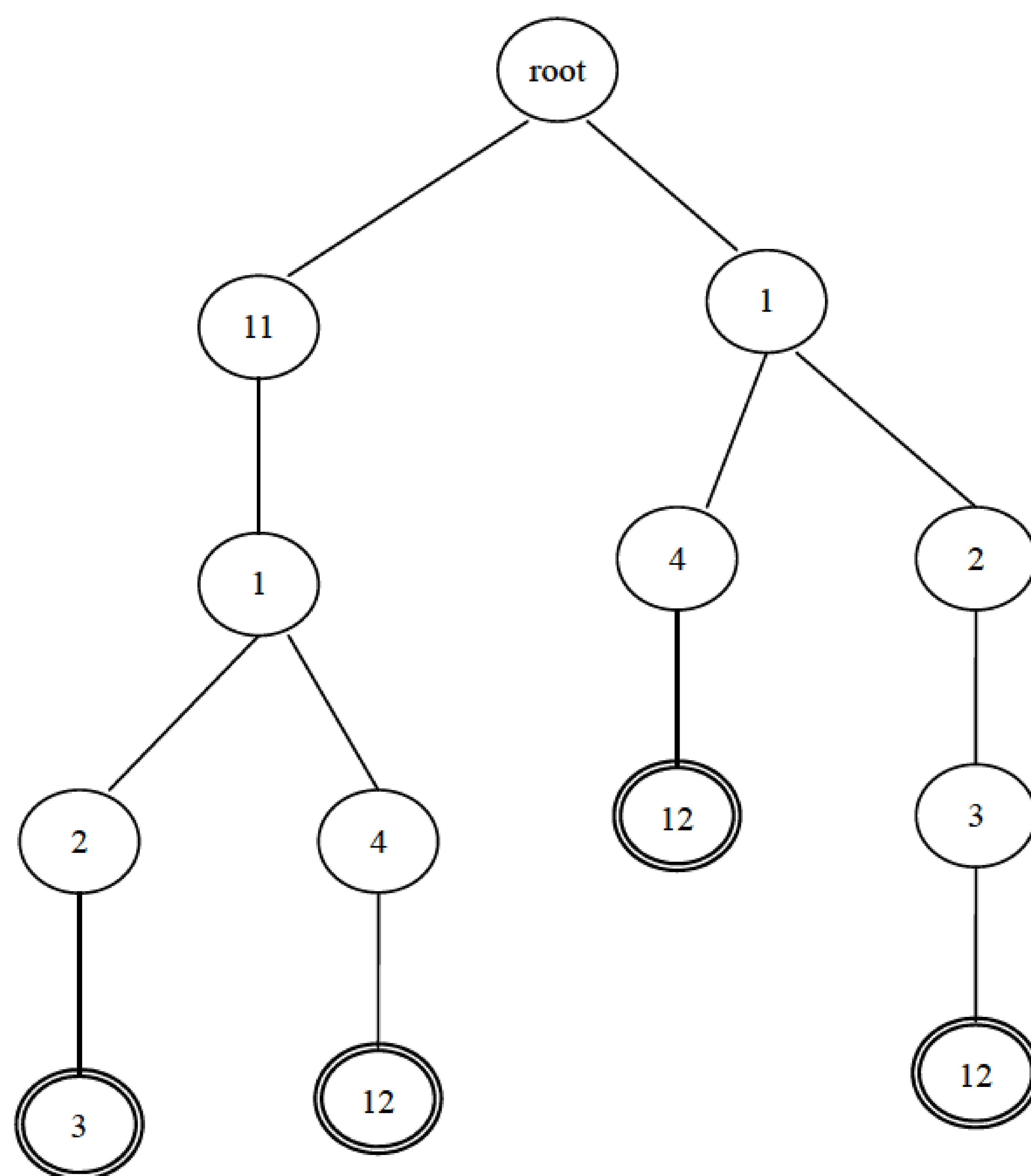


图 2-7　人名识别规则组成的标准检索树

一条规则中可能有多个人名，例如“李/nr 鹏/nr 和/d 江/nr 泽民/nr”对应一个复杂的规则：[1,4,13,1,2,3]。

定义标准检索树的节点：

```
public class TrieNode{
   private PersonType nodeKey;                    //键
   private ArrayList<NameSpan> nodeValue;         //值
   private boolean terminal;                      //标志这个节点是否可以结束的节点
   private Map<PersonType, TrieNode> children =
              new HashMap<PersonType, TrieNode>();  //引用到所有的孩子节点
}
```

所以可以通过匹配规则来识别未登录词。为了实现同时查找多个规则，可以把右边的模式组织成检索树，左边的模式作为节点属性。

NameSpan 用来指定一个区间，就是合并多长的未登录词语素成为一个未登录词。识别规则的左边部分就是一个 NameSpan 序列，而识别规则的右边部分就是一个 PersonType 序列。规则检索树的实现如下：

```
public class Trie {
   public TrieNode rootNode = new TrieNode(); //根节点
```

```
    //放入键/值对
    public void addProduct(ArrayList<PersonType> key, ArrayList<NameSpan> lhs)
{
        TrieNode currNode = rootNode;               //当前节点
        for (int i = 0; i < key.size(); ++i) {  //从前往后找键中的类型
          PersonType c = key.get(i);
          Map<PersonType, TrieNode> map = currNode.getChildren();
          currNode = map.get(c);                    //向下移动当前节点
          if (currNode==null) {
              currNode = new TrieNode();
              map.put(c, currNode);                 //孩子放入散列表
          }
        }
        currNode.setTerminal(true);                 //设置成可以结束的节点
        currNode.setNodeValue(lhs);                 //设置值
    }

    //根据键查找对应的值，也就是根据右边的 PersonType 序列看有没有对应的识别规则
    public ArrayList<NameSpan> find(ArrayList<PersonType> key) {
        TrieNode currNode = rootNode;               //当前节点
        for (int i = 0; i < key.size(); ++i) {  //从前往后找键中的类型
          PersonType c = key.get(i);
          currNode = currNode.getChildren().get(c);//向下移动当前节点
          if (currNode==null) {
              return null;
          }
        }
        if (currNode.isTerminal()) {                //是结束节点
          return currNode.getNodeValue();
        }
        return null;                                //没找到
    }
}
```

把规则加入到规则检索树的代码如下：

```
//构造规则的右部分：人名上文 姓氏 + 单人名
rhs = new ArrayList<PersonType>();
rhs.add(PersonType.preContext);     //人名上文
rhs.add(PersonType.surName);        //姓氏
rhs.add(PersonType.singleName);     //单名
//构造规则的左部分：人名上文之后是姓名
lhs = new ArrayList<NameSpan>();
lhs.add(new NameSpan(1, 2, PersonType.name)); //姓氏 和 单人名 组成完整的人名
rules.addProduct(rhs, lhs);         //把人名识别规则加入规则库
```

从特征词图中找规则检索树上可以匹配上的规则。也就是说，在特征词图上有一条路径正好也是可以在规则检索树上从开始走到结束节点。例如，图 2-8 中左边的特征词图状

态 0 接收输入“姓”以后转换到状态 1，状态 0 接收输入“上文”以后转换到状态 2。状态 1 和状态 2 被映射到右边的规则检索树，因为右边的规则检索树也存在从开始状态接收输入“姓”以后转换到一个新状态，从开始状态接收输入“上文”以后转换到另外一个新状态。

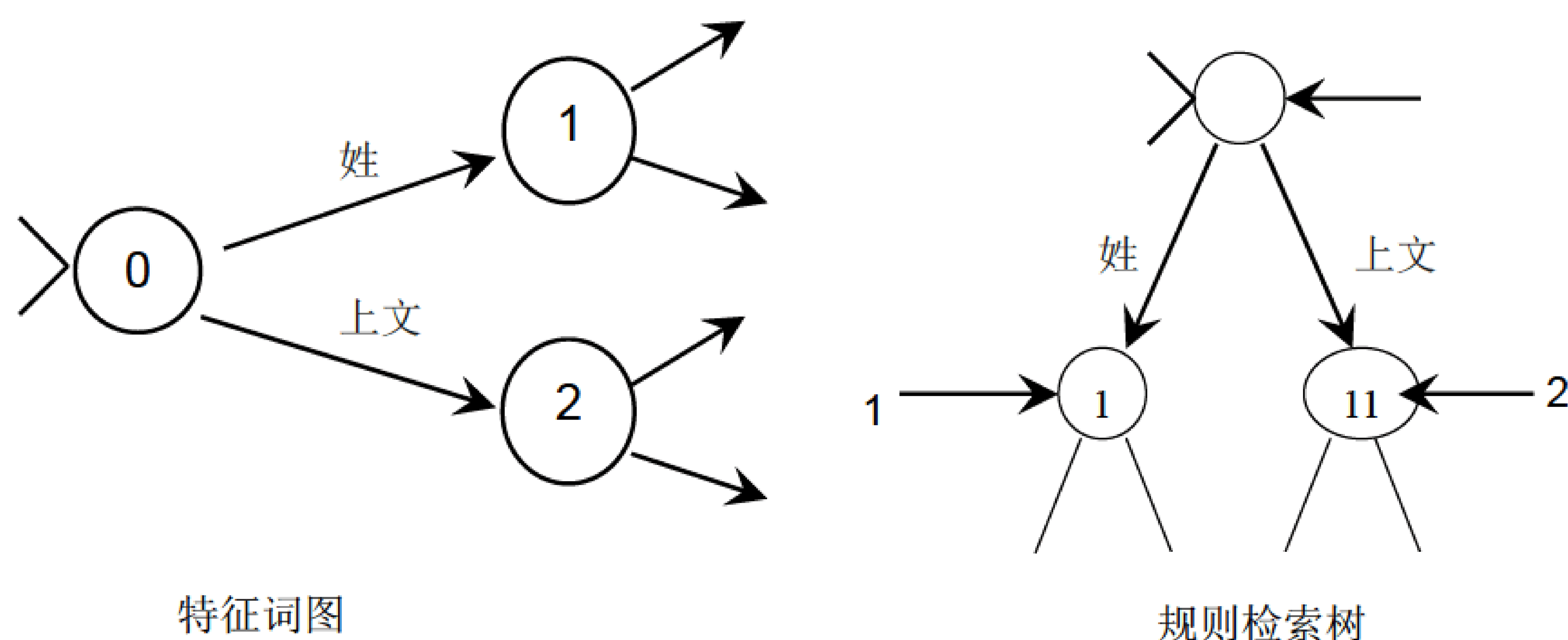


图 2-8　从人名特征词图上找匹配规则

把特征词图中的当前状态叫作 s_1，规则检索树的当前状态称为 s_2。状态 s_1 和 s_2 组成一个当前状态对（s_1, s_2）。例如，图 2-8 存在状态对（1,1）和（2,11）。

当前状态对 StatePair 类的部分代码如下：

```
public static class StatePair {
   int s1;  //特征词图中的当前状态编号
   TrieNode s2; //规则检索树的当前状态节点
}
```

在每个当前状态对中，都对状态 s_1 和 s_2 的所有可能接收的输入求交集。从特征词图找规则序列的每一步都要找输入交集，也就是求词图和规则树中的 PersonType 的交集。其实现代码如下：

```
public static class NextInput {  //有限状态自动机中的下一个输入
      int end;                   //词的结束位置，词图中下一个状态对的依据
      PersonType type;           //经过的类型，规则树中下一个状态对的依据
      String term;               //经过的词
}
```

UnknowGrammar 类的 intersection()方法的实现代码如下：

```
/**
 * 取得词图和规则树都可以向前进的步骤
 * @param edges  词图上的边
 * @param s 规则树上的类型
 * @return 共同的有效输入
 */
public ArrayList<NextInput> intersection(EntityLinkedList edges,
```

```
                                Set<PersonType> s) {
    ArrayList<NextInput> tmp = new ArrayList<NextInput>();
    for (EntityTokenInf x : edges) {
        if (x.data == null)
            continue;
        for (EntityTypes.EntityTypeInf typeInf : x.data) {
            if (s.contains(typeInf.pos)) {  //规则树上的类型包含词所属的类型
                tmp.add(new NextInput(x.end, typeInf.pos, x.termText));
            }
        }
    }
    return tmp;
}
```

找出人名相关的序列，也就是把词图映射到检索树上。

```
public static class MatchValue {
    ArrayList<NameSpan> left;      //规则的左边部分
    ArrayList<PersonType> right;   //规则的右边部分
    ArrayList<String> term;        //对应的词序列
}
```

词图中的每个节点都可能有几条路径通过，但只保留那些能走到底的路。查找过程的输入是特征词图开始找的位置，返回多个可能的识别规则的 UnknowGrammar.intersect()方法实现如下：

```
/**
 * 词图映射到检索树上，也就是从词图指定位置开始找识别规则
 * @param g 人名特征词图
 * @param offset 开始位置
 * @return  匹配结果
 */
public ArrayList<MatchValue> intersect(AdjList g, int offset) {
    ArrayList<MatchValue> match = new ArrayList<MatchValue>(); //映射结果
    Stack<StatePair> stack = new Stack<StatePair>(); //存储遍历状态的堆栈
    ArrayList<PersonType> path = new ArrayList<PersonType>();  //类型序列
    ArrayList<String> term = new ArrayList<String>();  //人名特征词序列

    stack.add(new StatePair(path, offset, rules.rootNode, term));
    while (!stack.isEmpty()) { //堆栈内容不是空
        StatePair stackValue = stack.pop(); //弹出堆栈

        //取出图中当前节点对应的边
        EntityLinkedList edges = g.edges(stackValue.s1);

        //取出树中当前节点对应的类型
        Set<PersonType> types = stackValue.s2.getChildren().keySet();
        ArrayList<NextInput> ret = intersection(edges, types);
        if (ret == null)
```

```
            continue;
        for (NextInput edge : ret) { //遍历每个有效的输入
            //向下遍历树
            TrieNode state2 = stackValue.s2.getChildren().get(edge.type);
            //向前遍历图上的边
            int end = edge.end;
            if (state2 != null) {
                ArrayList<PersonType> p = new ArrayList<PersonType>(
                        stackValue.path);
                p.add(edge.type);

                ArrayList<String> t = new ArrayList<String>(stackValue.term);
                t.add(edge.term);

                stack.add(new StatePair(p, end, state2, t)); //压入堆栈
                if (state2.isTerminal()) { //是可以结束的节点
                    match.add(new MatchValue(state2.getNodeValue(), p, t));
                }
            }
        }
    }

    return match;
}
```

特征词图的每个节点开始向后找规则。

```
UnknowGrammar unknowGrammar = UnknowGrammar.getInstance();

for (int i = 0; i < atomCount; ++i) {
    //从特征词图指定位置开始求交集
    ArrayList<MatchValue> match = unknowGrammar.intersect(g, i);
    //处理找到的未登录词
}
```

为"程正泰的父亲是一位京剧票友，素与杨宝森先生交好。"准备识别模板：

```
UnknowGrammar g = new UnknowGrammar();
String right =
"<Begin><surName>{surName}<doubleName1>{doubleName1}<doubleName2>{doubleN
ame2}的父亲是一位";                                //匹配人名的规则
String handlerName = "PersonNamesd1d2";         //处理器名
g.add(handlerName, right);                      //人名处理器
```

表示文本中的人名的 PersonToken 类定义如下：

```
public class PersonToken {
    public String person;                       //人名
    public int start;                           //开始位置

    public PersonToken(String person, int start) {
        this.person = person;
```

```
            this.start = start;
        }
    }
```

存储提取参数的 PairListPersonToken 类：

```
public class PairListPersonToken {
   List<Map.Entry<String,PersonToken>> args;  //存储类型和对应的 PersonToken

   public PairListPersonToken() {
      args = new ArrayList<Map.Entry<String,PersonToken>>();
   }

   public void add(String k,PersonToken v){   //增加键和对应的值
      Map.Entry<String,PersonToken> entry =
              new AbstractMap.SimpleEntry<String, PersonToken>(k, v);
      args.add(entry);
   }

   public PersonToken getFirst(String key){   //取得第一个键对应的值
      for(Map.Entry<String,PersonToken> e:args) {
        if(e.getKey().equals(key)) {
            return e.getValue();
        }
      }
      return null;
   }
}
```

提取人名的代码如下：

```
String text ="程正泰的父亲是一位京剧票友，素与杨宝森先生交好。";      //文法提取器
GrammarExtractor unknowFind = new GrammarExtractor();
List<PersonToken> result = unknowFind.extract(text);          //提取结果
System.out.println(result);
```

2.8　文本归一化

为了把日期和事件联系起来，可以归一化日期。

例如：

```
King George VI of England died on Feb 6, 1952.
```

可以转换成：

```
King George VI of England died on February 6, 1952.
```

归一化是识别数字、缩写、首字母缩略词和惯用语的过程，并根据需要将它们转换为全文。

想要识别的绝对日期将采用以下模式的规范化形式：

- 18 Feb 1997 → 1997/02/18
- 20th of July → XXXX/07/20
- 1992 → 1992

2.9 依存树模型

可以使用依存语言模型来建模单词之间的句法依赖关系。

使用依存文法构建依存语言模型。依存文法认为词之间的关系是有方向的，通常是一个词支配另一个词，这种支配与被支配的关系就称作依存关系，而且包括汉语和英语的大多数语言满足投射性。所谓投射性是指：如果词 p 依存于词 q，那么 p 和 q 之间的任意词 r 就不能依存到 p 和 q 所构成的跨度之外。

例如，汉语句子“这是一本书。”的依存文法结构如图 2-9 所示。

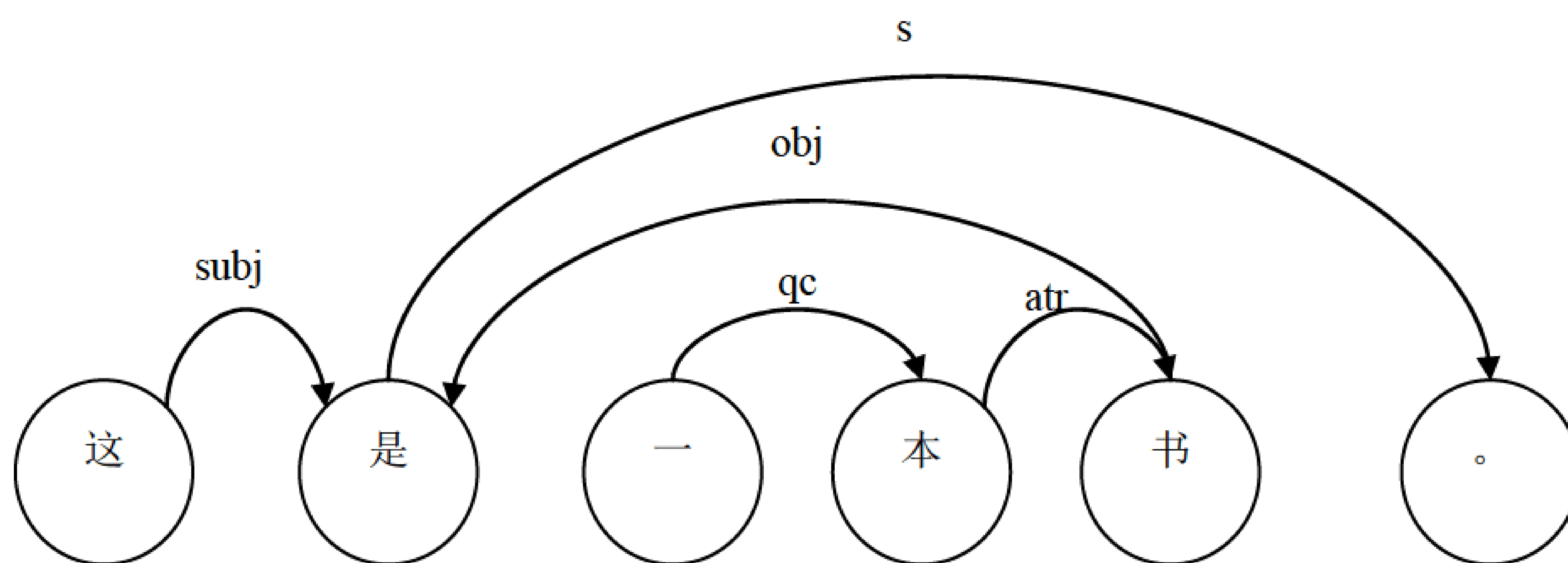


图 2-9 “这是一本书。”的依存文法结构图

图 2-9 中带箭头的弧的起点为从属词，箭头指向的是支配词，弧上标记为依存关系标记。例如，句号“。”支配“是”；动词“是”是句子的谓语，它支配主语“这”和宾语“书”，故“是”是支配词，“这”和“书”是从属词；“s”“subj”“obj”是依存关系标记。支配词也叫核心词，从属词也叫修饰词。

数词“一”作量词“本”的量词补足语，“本”是支配词，“一”是从属词，“qc”是依存关系标记。数量短语“一本”作名词“书”的定语，名词“书”支配量词“本”，“atr”是依存关系标记。

依存文法也可以表示成图 2-10 这样的树结构。因为总是连接线下面的词依赖上面的词，所以图 2-10 中的箭头可以省略。

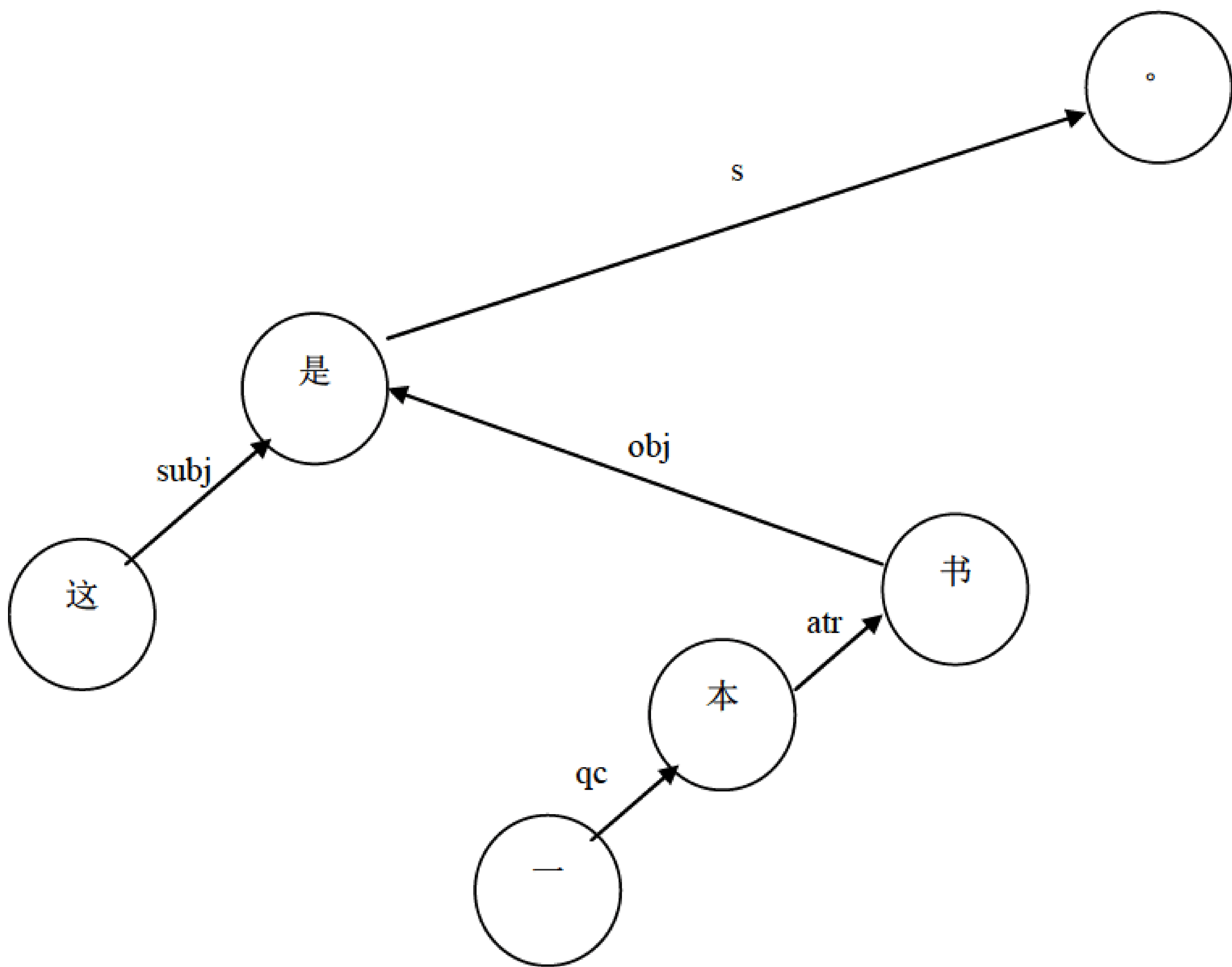


图 2-10　“这是一本书。”的依存文法树

在依存语言模型中利用依存树有很多可能的方法。

构造依存语言模型的最简单方法是使用依存树的拓扑结构 T。每个单词都由其父亲调节。例如图 2-11 所示句子的概率计算公式如下：

```
    P (s | T )= P ( the | boy) P ( boy | find) P(will| find ) P (find | <NONE> )
P ( it | find )P (interesting |find)
```

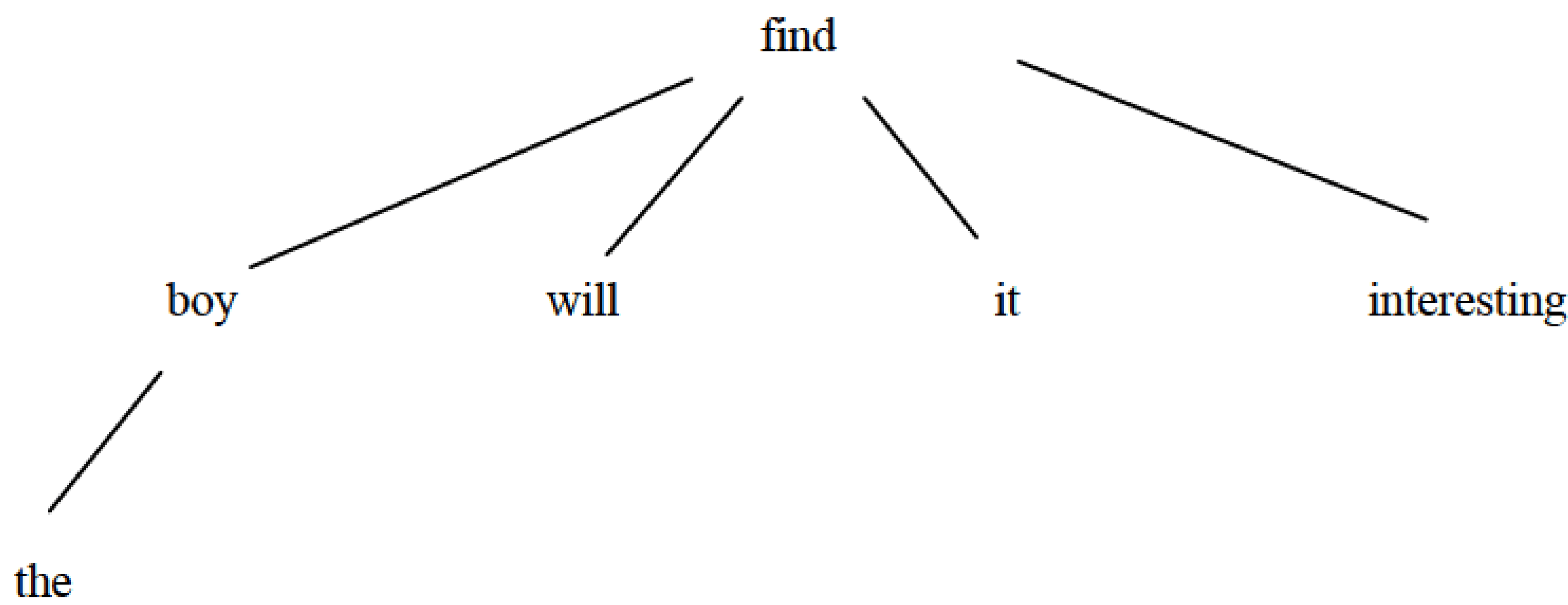


图 2-11　句子“the boy will find it interesting”的依存树

2.10　情感分析

人们常常会对某个事物（如产品）发表自己的看法或评论，计算机可以判断该看法

或评论是属于对该事物的积极或消极意见。这就是文本倾向性分析。可以结合核磁共振，通过对大脑中的兴奋区域成像来分析“喜欢”或“厌恶”等情感倾向。这里讨论的文本倾向性分析，英文称为 sentiment analysis 或 opinion mining。基本的目标就是实现区分出正面、负面或者中性，这称为极性分类（polarity classification）。可以按好恶程度分出更多的级别，例如 1~5 星级，这称为星级评分（multi-way scale）。

有文档级别的情感识别，例如对某个电影或酒店的评论自动分类出极性或者星级，这样区分出好评和差评。也许想进一步对好在哪里，差在何处做更细致的分析，所以出现了更细粒度的基于特征的情感识别。例如，区分出对手机的屏幕或者照相机的画质的评价。为了准确地识别级性，可以考虑对文本的主客观语句分类，提取出 n 个最主观的句子来概括整个评论的褒贬倾向。从技术上来说，就是从主客观混合文本语料中抽取表示主观性的文本。

为了实现基于特征的情感识别，需要从上下文提取出评价的对象，需要提取描述对象的特征，然后判断倾向性描述在每个特征上的极性。“特征”一词在这里既表示描述对象的组成，也表示属性。

特征抽取是获得关于主题某一方面的具体描述，如汽车的油耗与操控性，数码相机的电池寿命。和信息抽取相比，情感分析中的特征抽取更加自由，因为获得的结果不要求是结构化的。在某些应用中，特征抽取比情感取向判断更加重要，因为需要关注用户的具体意见。例如对某款手机的评价统计：

```
手机:
褒义: 125 <独立的评价句子>
贬义: 7 <独立的评价句子>
特征: 续航能力
褒义: 123 <独立的评价句子>
贬义: 6 <独立的评价句子>
特征: 大小
褒义: 82 <独立的评价句子>
贬义: 10 <独立的评价句子>
```

对事物的观点有直接观点和对比观点两种：

- 直接观点（direct opinion）：例如，这款手机的画质的确有点烂。
- 对比观点（comparative opinion）：例如，这款手机的画质比 iPhone-x 好。进行这类情感分析时，首先要确定观点的目标对象是谁。在这个例子中需要用到指代消解确定这款手机指哪款手机。

有时候，作者把情感和事实一起来表达，例如“即便是面对强逆光，iPhone-x 的表现也堪称专业”。也就是说，情感和具体的特征是分不开的。

除了这些经典的问题外，在针对社会媒体的情感分析中，我们面临更多的挑战。例

如，并非所有以与主题相关的用户为中心的内容都是重要的，而其中只有少部分能引起关注和讨论，甚至进而影响其他用户的观念和行为。因此，评估它们的影响力和预测它们是否得到关注具有重要的应用价值。

除此以外，不合理地利用社会媒体的影响力也值得关注。制造事端打击竞争对手，恶作剧心理造谣生事，收受商家好处为特定产品夸大宣传，是典型的误导公众行为。

首先从文本中抽取描述对象的特征。例如，针对汽车的用户体验信息，关于操控性、舒适性、油耗、内饰、配置等方面的评价等被分别抽取列出，因此可以收集到不同用户关于同一特征的描述并在不同品牌、不同时间段、不同用户群的范围内统计加以比较评估，这样的数据能直接地、准确地反映用户的消费情况和市场反应。其次，需要评估一个用户言论的内在价值和预测将来的关注度。从实务操作上来说，有些重要的言论和事件在几小时内就会引起广泛的关注。相关的厂家可以及时发现和跟进这种对其产品销售和品牌形象具有重要影响的言论。

为了获取标注好的文本倾向，可以从评论网站，比如 Booking.com 等网站抓取所有的酒店评论，这些评论用星级评价来代表褒贬度。

常见具有语义倾向词语的词性及示例如表 2-4 所示。

表 2-4　有语义倾向词语的词性表

词 性 编 码	词　　性	示　　例
a	形容词	美丽、丑陋
n	名词	英雄、熊市、粉丝、流氓
v	动词	发扬、贬低
d	副词	昂然、暗地
i	成语	宾至如归、叶公好龙
p	习惯语	双喜临门、顺竿爬

事实上，对一篇文章而言，它所表达的情感的正面或负面是通过主观语句体现出来的，如“产品质量好!”但是像“它的售价刚好是￥50 元!”这样的客观语句，虽然有“好”这一特征词，但并不表达任何情感。但是如果能区分一篇文章中的主观语句和客观语句，只对主观语句进行特征选择，会对分类的准确率有很大提高。

观点搜索系统使得用户能够查找关于一个对象的评价观点。典型的观点搜索查询包括以下两种类型：

- 搜索关于一个特定对象或对象特征的观点。搜索用户只要简单给出对象和/或对象的特征即可。
- 搜索一个人或组织关于一个特定对象或者对象特征的观点。用户需要给出观点拥有者的名字和对象的名字。

判断用户的情感取向（polarity）是喜欢、不喜欢还是中性的，可以通过对大量用户的感情取向进行统计，进而了解用户对特定产品的好恶，甚至对具体的某个特征（如数码相机的镜头、电池寿命等）作出直接的判断和比较。

开源机器学习框架 ML.NET(https://github.com/dotnet/machinelearning)包含了情感识别的实现。

要开始使用 ML.NET，请从包管理器安装 ML.NET 的 NuGet 包：

```
Install-Package Microsoft.ML
```

用于训练模型以预测文本样本中的情绪的代码如下：

```
var dataPath = "sentiment.csv";
var mlContext = new MLContext();
var loader = mlContext.Data.CreateTextLoader(new[]
   {
      new TextLoader.Column("SentimentText", DataKind.String, 1),
      new TextLoader.Column("Label", DataKind.Boolean, 0),
   },
   hasHeader: true,
   separatorChar: ',');
var data = loader.Load(dataPath);
var learningPipeline = mlContext.Transforms.Text.FeaturizeText("Features",
                                    "SentimentText")
      .Append(mlContext.BinaryClassification.Trainers.FastTree());
var model = learningPipeline.Fit(data);
现在从训练出的模型我们可以做出预测：
var predictionEngine =
     model.CreatePredictionEngine<SentimentData,
SentimentPrediction>(mlContext);
var prediction = predictionEngine.Predict(new SentimentData
{
    SentimentText = "Today is a great day!"
});
Console.WriteLine("prediction: " + prediction.Prediction);
```

如果需要采用 Java 实现，则 Stanford Core NLP(https://github.com/stanfordnlp/CoreNLP)中包含现成的实现。

2.11　本章小结

词性标注是简单而有用的语言学分析过程。例如，可以提取文本中的名词用作文本分类的依据。有很多常用词有多个可能的词性。早期的词性标注往往使用 HMM 这样的纯概率的方法来标注词性，后续增加了语法和语义知识来帮助提高词性标注的准确性。

第 3 章
搜索引擎听懂语音

可以根据一段语音中的频率变化来切分语音，但由于语音是许多不同频率信号的混合，因此从频谱而不是单一频率考虑可能更有成果。即使对于固定音高的持续音符，除了音符的基本频率之外，还会出现大量的泛音和谐波。而对于实际的语音，由于元音和辅音的不同音调特性，即使在短片段内频谱也会剧烈变化。

语音识别技术，也被称为自动语音识别（automatic speech recognition，ASR）。它是一种交叉学科，与人们的生活、工作和学习密切相关。其目标是将说话者的词汇内容转换为计算机可读的输入，例如按键、二进制编码或者字符序列。比如，将来打银行的客服电话，可以直接和银行系统用口语对话，而不是“普通话请按 1”这样把人当成机器的询问，实现语音交互。

在通信中，可以把对方的语音留言转换成文字。进一步地，还可以根据识别出的文字识别语意。这样，可以让机器和人交流。例如：儿童识别图片后，可以说出这个图是老虎还是大象；系统使用语音识别技术判断孩子的回答是否正确；对于不正确的，系统自动给出提示。

开放式语音识别不容易做好，可以辅助人工输入字幕，类似语音输入法。

3.1 语音识别总体结构

语音识别可以被看作广义上的标注问题。给定声学输出 $A_{1,T}$(由一个声学事件的序列

组成 $a_1,\cdots,a_T$)，需要找到单词序列 $W_{1,R}$ 最大化概率：

$$\arg\max_{W} P\left(W_{1,R} \mid A_{1,T}\right)$$

根据贝叶斯公式重写这个公式，然后删除在通过比较大小找最大值的过程中没有意义的分母，把问题转换成计算：

$$\arg\max_{W} P\left(A_{1,T} \mid W_{1,R}\right) P\left(W_{1,R}\right)$$

这里把 $P(W_{1,R})$叫作语言模型，而 $P(A_{1,T}|W_{1,R})$叫作声学模型。语音识别过程如图 3-1 所示。

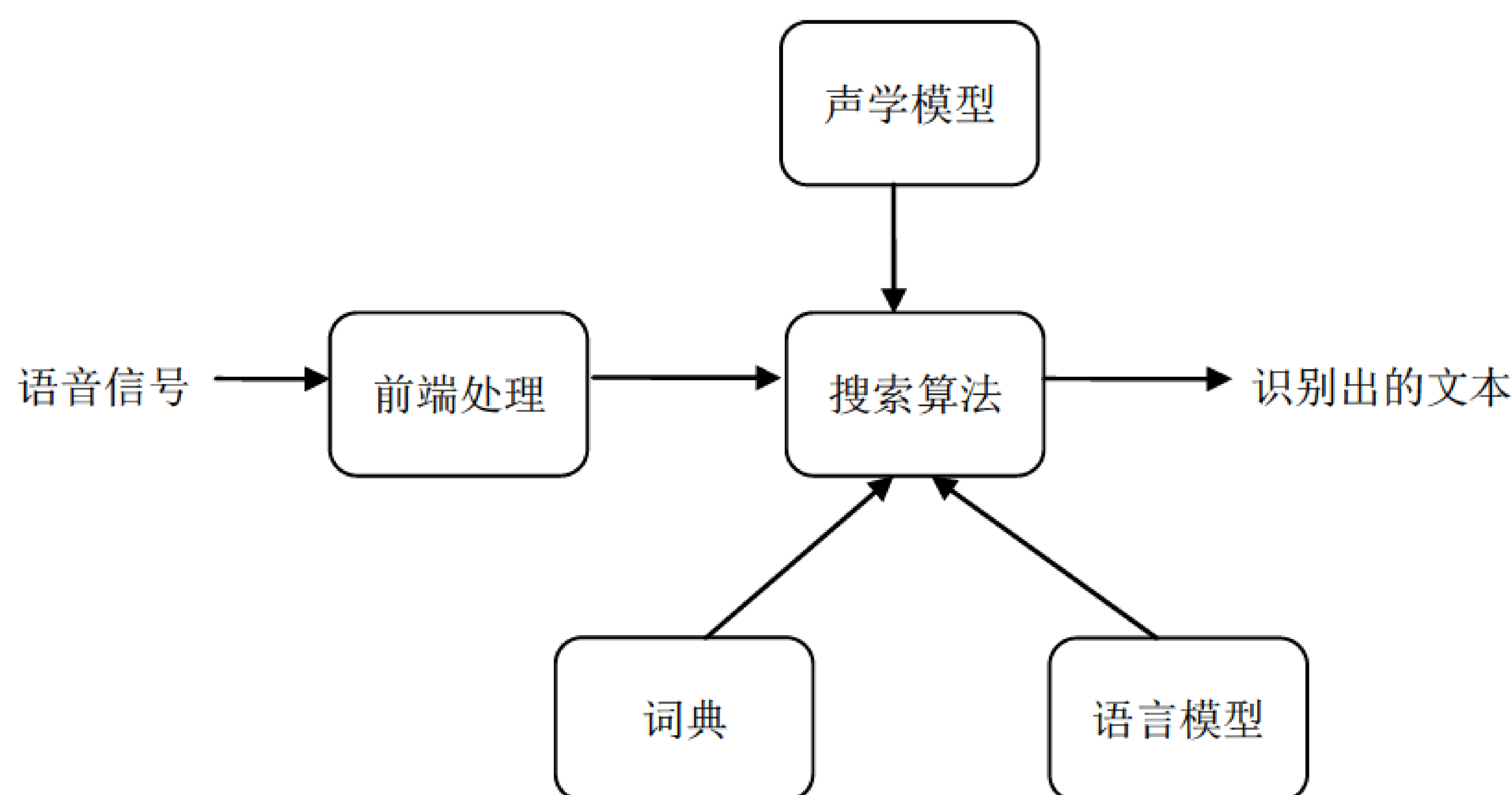


图 3-1 语音识别过程

人类获得信息的 80%来自于图像。图像信息具有传递速度快、信息量大等一系列特点，但语音识别在车载系统、智能音响等领域也有非常关键的应用。

为了能开发出有效的语音识别系统，2009 年 Kaldi 在约翰霍普金斯大学诞生了。Kaldi 不是一个语音识别系统，它是一个建立语音识别系统的系统。Kaldi 使用运行于 Linux 操作系统的 C++、Perl、Python、Bash 等多种语言开发。接下来介绍需要用到的 Linux 基础知识。

3.2 Kaldi 快速入门

首先介绍如何安装和使用 Kaldi。

3.2.1　安装 Kaldi

一般在 Linux 下运行 Kaldi。介绍在 Linux 下安装 Kaldi 的过程。

首先安装 git：

```
# yum install git
```

然后下载 kaldi：

```
# git clone https://github.com/kaldi-asr/kaldi.git kaldi --origin upstream
```

可以参考下载文件中的说明进行安装。在源代码的根目录下有一个 INSTALL 文件，其中描述了安装步骤。首先在 tools/下查看 INSTALL 安装指令；然后在 src/下查看 INSTALL 安装指令。

到/tools/extras 目录下，运行 check_dependencies.sh 脚本，检查安装过程中所依赖的工具是否存在。运行的结果会提示安装依赖软件的命令。

首先在./tool 目录下编译源代码，然后在./src 目录下编译。

在./tool 目录下输入 make 命令就可以编译，输入 make -j 4 命令可以用多核并行处理的方式加快速度。

之后切换到./src 目录下，运行如下命令：

```
./configure
make depend
make -j 4
```

为了方便以后使用，可以把环境打包：

```
tar -czf kaldiLinux.tar.gz ./kaldi
```

egs 目录下保存着一些指定数据集上的训练步骤（shell 脚本）以及测试的结果。最简单的是 yesno 例子。

3.2.2　yesno 例子

首先运行下面这个例子：

```
# cd ./egs/yesno/s5
#./run.sh
```

经过一段时间的训练和测试，可以看到运行结果。

```
%WER 0.00 [ 0 / 232, 0 ins, 0 del, 0 sub ] exp/mono0a/decode_test_yesno/wer_10
```

这里的 WER（word error rate）是字错误率，是一个衡量语音识别系统的准确程度的度量。其计算公式是 $WER=(I+D+S)/N$，其中 I 代表被插入的单词个数；D 代表被删除的单词个数；S 代表被替换的单词个数。也就是说，把在识别出来的结果中多认的、少认的、认错的全都加起来，再除以总单词数。这个数字当然是越低越好。

这里的 WER 为 0.00。说明全部识别正确。

数据集有 62 个.wav 文件，采样频率为 8kHz。所有音频文件由 Kaldi 项目的匿名男性贡献者记录，并包含在项目中用于测试目的。我们把它们放在 wave_yesno 目录中，但数据集也可以在 http://openslr.org/resources/1/waves_yesno.tar.gz 找到。在每个文件中，这个人说 8 个词，每个单词都是“ken”或“lo”（希伯来语中的“是”和“否”），因此每个文件都是 8 个是或否的随机序列。文件名称表示单词序列，1 表示是，0 表示否。

```
waves_yesno/1_0_1_1_1_0_1_0.wav
waves_yesno/0_1_1_0_0_1_1_0.wav
...
```

这就是所有的原始数据。现在将这些.wav 文件变成 Kaldi 可以读取的数据格式。

3.2.3 数据准备

将 62 个波形文件大致分为两半：31 个用于训练，其余用于测试。

创建数据目录，然后在其中有 train_yesno 和 test_yesno 两个子目录。

使用一个称为 data_prep.py 的 Python 脚本生成必要的输入文件。

读取 wave_yesno 中的文件列表。

生成两个列表，一个存储以 0 开头的文件的名称，另一个存储以 1 开头的文件的名称，忽略其余的文件。

对于每个数据集（训练集和测试集）都需要生成代表原始数据的文件——音频和讲稿。

1. 讲稿文件

每行一句话，格式是：

```
<utt_id> <transcript>
```

例如： 0_0_1_1_1_1_0_0 NO NO YES YES YES YES NO NO

现在使用没有扩展名的文件名作为 utt_ids。

虽然录音是希伯来语，这里使用英语单词 YES 和 NO 代替，以避免使问题复杂化。

2. wav.scp

唯一 ID 的索引文件。

```
<file_id> <带路径的 wave 文件名或者获取 wav 文件的命令>
```

例如： 0_1_0_0_1_0_1_1 waves_yesno/0_1_0_0_1_0_1_1.wav

这里再次使用文件名作为文件 ID。

3. utt2spk

对于每句话，标记是哪个说话者说出来的。

```
<utt_id> <speaker_id>
```

例如：0_0_1_0_1_0_1_1 global

由于这个例子中只有一个说话者，所以让我们使用“global”作为说话者标识。

4. spk2utt

- 简单的反向索引 utt2spk (<speaker_id> <all_hier_utterences>)。
- 可以使用 Kaldi 工具程序生成。
- utils/utt2spk_to_spk2utt.pl data/train_yesno/utt2spk > data/train_yesno/spk2utt。

数据目录看起来像这样：

```
data
├───────train_yesno
│    ├───────text
│    ├───────utt2spk
│    ├───────spk2utt
│    └───────wav.scp
└───────test_yesno
    ├───────text
    ├───────utt2spk
    ├───────spk2utt
    └───────wav.scp
```

3.2.4　词典准备

本部分将介绍如何为 Kaldi 识别器构建语言知识——词典和音素词典。

接下来建立词典。先从根目录创建中间的 dict 目录开始。

```
mkdir dict
```

在这种玩具语言中，只有两个词：YES 和 NO。为了简单起见，我们假设它们是一个单音素词：Y 和 N。

```
echo -e "Y\nN" > dict/phones.txt                #音素词典
echo -e "YES Y\nNO N" > dict/lexicon.txt       #单词发音词典
```

然而，在真实的讲话中，不仅有表达语言的人的声音，还有沉默和噪声。

Kaldi 把所有这些非语言的声音称为“沉默”。例如，即使在这个小而受控制的录音中，也在每个单词之间的暂停。因此，需要一个额外的音素“SIL”代表沉默。而且可以在所有单词的结尾发生。Kaldi 称这种沉默为“可选的”。

```
echo "SIL" > dict/silence_phones.txt
echo "SIL" > dict/optional_silence.txt
mv dict/phones.txt dict/nonsilence_phones.txt
```

现在修改词典以包含沉默。

```
cp dict/lexicon.txt dict/lexicon_words.txt
echo "<SIL> SIL" >> dict/lexicon.txt
```

请注意，“<SIL>”也将用作我们的 OOV 词。

dict 目录应该最终得到这 5 个文件：

- lexicon.txt：词素-音素对的完整列表。
- lexicon_words.txt：单词-音素对列表。
- silence_phones.txt：无声音素列表。
- nonsilence_phones.txt：非无声音素列表。
- optional_silence.txt：可选无声音素列表（这看起来和 silence_phones.txt 相同）。

最后，需要将我们的字典转换为 Kaldi 能接受的数据结构——有限状态转换机（FST）。在 Kaldi 提供的许多脚本中，我们将使用 utils/prepare_lang.sh 生成 FST 就绪的数据格式来表示我们的语言定义。

```
utils/prepare_lang.sh --position-dependent-phones false <RAW_DICT_PATH>
<OOV> <TEMP_DIR> <OUTPUT_DIR>
```

我们正在使用**--position-dependent-phones** 参数值是假的——在我们的小小的玩具语言中。毕竟没有足够的上下文。对于所需参数，我们将使用：

- <RAW_DICT_PATH>：dict。
- <OOV>："<SIL>"。
- <TEMP_DIR>：可以在任何地方。这里在 dict 里面建一个新的目录 tmp。
- <OUTPUT_DIR>：此输出将用于进一步的训练。将其设置为 data/lang。

我们给出了一个用于 yesno 数据的样例一元语法模型。你会在 lm 目录下找到一个 ARPA 格式的语法模型。然而，语言模型也需要转换成 FST。为此，Kaldi 也带来了许多程序。在这个例子中，我们将使用绑定的脚本 lm/prepare_lm.sh。它将生成正确格式的 LM FST 并将其放入 data/lang_test_tg 中。

接下来是 MFCC 特征提取和训练 GMM 模型。

首先提取梅尔频率倒谱系数。

```
steps/make_mfcc.sh --nj <N> <INPUT_DIR> <OUTPUT_DIR>
```

- --nj <N>：处理器数量，默认为 4。
- <INPUT_DIR>：放训练集数据的地方。
- <OUTPUT_DIR>：让我们输出到 exp/make_mfcc/train_yesno，遵循 Kaldi 配方惯例。

现在归一化倒谱特征。

```
steps/compute_cmvn_stats.sh <INPUT_DIR> <OUTPUT_DIR>
```

<INPUT_DIR>和<OUTPUT_DIR>与上述相同。

这些 shell 脚本（.sh）都是通过 Kaldi 二进制文件的管道操作，它们都是些文本处理。要查看实际执行了哪些命令，请参阅<OUTPUT_DIR>中的日志文件，或者最好看脚本内容。

我们将训练单音素模型，因为我们假设，在我们的玩具语言中，音素不依赖于上下文。（这当然是一个荒谬的假设）

```
    steps/train_mono.sh  --nj  <N>  --cmd  <MAIN_CMD>  <DATA_DIR>  <LANG_DIR>
<OUTPUT_DIR>
```

- --cmd <MAIN_CMD>：因为要使用本地机器资源，所以使用“utils/run.pl”管道。
- --nj <N>：来自说话人的发言不能并行处理。由于我们只有一个，我们只能使用 1 个任务。
- <DATA_DIR>：训练数据的路径。
- <LANG_DIR>：语言定义的路径(prepare_lang 脚本的输出)。
- <OUTPUT_DIR>：和前面一样，使用 exp/mono。

这将产生用于声学模型的基于 FST 的词图。Kaldi 提供了一个查看模型内部的工具（现在可能还没有任何意义）。

```
    /path/to/kaldi/src/fstbin/fstcopy 'ark:gunzip -c exp/mono/fsts.1.gz|' ark,
t:- | head -n 20
```

这将以人可读（!!）的格式打印出前 20 行词图（每列表示：Q-from，Q-to，S-in，S-out，Cost）。

现在我们完成了声学模型的训练，接下来是图解码。对于解码，我们需要一个新的输入，通过我们的 AM&LM 词图。我们为 data/test_yesno 准备了单独的测试集。现在是时候将其投影到特征空间了，使用 steps/make_mfcc.sh 和 steps/compute_cmvn_stats.sh。

然后，需要建立一个完全连接的 FST 网络。

```
    utils/mkgraph.sh --mono data/lang_test_tg exp/mono exp/mono/graph_tgpr
```

这将在 exp/mono/graph_tgpr 目录中构建一个连接的 HCLG。

最后，需要使用解码脚本找到测试集中话语的最佳路径。查看解码脚本，找出什么要作为它的参数，然后运行它。将解码结果写入 exp/mono/decode_test_yesno。

```
    steps/decode.sh
```

这将产生输出目录中的 lat.N.gz 文件，其中 N 从 1 增加到用户使用的作业数（对目前这个任务来说，必须为 1）。这些文件包含由用户的解码操作的第 N 个线程处理的发言的词图。

请参阅 exp/mono/decode_test_yesno/wer_X 文件以查看 WER，参阅 exp/mono/decode_test_yesno/scoring/X.tra 查看讲稿。这里 X 表示语言模型权重、LMWT、每次迭代使用的评分脚本，将 lat.N.gz 文件中的话语的最佳路径解释为单词序列。（记住 N 在 decoing 操

作期间#thread）如果需要，可以在调用 score.sh 时，使用--min_lmwt 和--max_lmwt 选项来特意指定权重。

或者如果有兴趣获取每个重新编码文件的单词级对齐信息，请查看 steps/get_ctm.sh 脚本。

3.2.5 构建一个简单的 ASR

将数据分为训练和测试集，建立一个 ASR 系统，对其进行训练，测试并获得一些解码结果。

开始在 kaldi/egs 目录中创建一个 digits 文件夹。把所有与你的项目相关的东西都放在这里。

首先准备音频数据和语言数据。

这里假设您想要建立一个基于您自己的音频数据的 ASR 系统。例如，100 个文件的数据集。文件格式是 WAV。每个文件包含 3 个以英文记录的口语数字。这些音频文件中的每一个以可识别的方式命名（例如 1_5_6.wav，在我的模式中这个语音句子是“一、五、六”），并且在特定记录会话期间放置在表示特定说话人的可识别的文件夹中。可能有一种情况，有同一人的录音，但在两个不同的质量/噪声环境中，这时候要将它们放在单独的文件夹中。所以总结一下，示范数据集看起来像这样：

- 10 个不同的说话人（ASR 系统必须对不同的说话人进行训练和测试，所以说话的人越多越好）。
- 每位说话人说 10 句话，100 个句子/话语（100 个 *.wav 文件中放置在与特定说话人有关的 10 个文件夹中，即每个文件夹中有 10 个*.wav 文件）。
- 300 个词（从零到九的数字），每个句子/话语由 3 个词组成。

无论您的第一个数据集是什么，请根据您的具体情况调整这个例子。小心大数据集和复杂的语法，所以从简单的东西开始。在这种情况下只包含数字的句子不错。

到 kaldi-trunk/egs/digits 目录并创建 digits_audio 文件夹。在 kaldi-trunk/egs/digits/digits_audio 中创建两个文件夹：train 和 test。选择一个您选择的说话人来表示测试数据集。使用该说话人的“speakerID”作为 kaldi-trunk/egs/digits/digits_audio/test 目录中另一个新文件夹的名称。然后把所有与该人有关的音频文件放在一起。将其余（9 个说话人）的音频文件放入 train 文件夹，这将是您的训练数据集。还可以为每个说话人创建子文件夹。

首先准备声学数据。必须创建一些文本文件，用来允许 Kaldi 与您的音频数据打交道。将这些文件看成必须完成的。您将在此部分（以及语言数据部分）中创建的每个文件都可以被视为具有一定数量的字符串（每个字符串在一个独立行）的文本文件。这些

字符串需要排序。如果您遇到任何排序问题，可以使用 Kaldi 脚本检查（utils/validate_data_dir.sh）和修复（utils/fix_data_dir.sh）数据顺序，并且为了您的信息，utils 目录将在工具附件部分附加到您的项目。

在 kaldi-trunk/egs/digits 目录中，创建一个文件夹数据。然后在里面创建 test 和 train 子文件夹。在每个子文件夹中创建以下文件（因此，在 test 和 train 子文件夹中以同样的方式命名文件，只不过是与之前创建的两个不同的数据集相关）：

（1）spk2gender。该文件记录说话人的性别。如我们假设的，“speakerID”是每个说话人的唯一名称（在这种情况下，它也是一个“recordingID”）——每个说话人只有一个录音会话中的一个音频数据文件夹）。在这个例子中，有 5 名女性和 5 名男性说话者（f =女性，m =男性）。

模式：<speakerID> <gender>

例如：

```
    cristine f
    dad m
    josh m
    july f
    #等等...
```

（2）wav.scp。该文件用相关的音频文件连接每个发言（在特定记录会话期间由一个人说的句子）。如果坚持我的命名方法，'utteranceID'只不过是'speakerID'（说话人的文件夹名称），附带了*.wav 文件名，而没有'.wav'结尾（看下面的例子）。

模式：<uterranceID> <full_path_to_audio_file>

例如：

```
    dad 4 4 2 /home/{user}/kaldi-trunk/egs/digits/digits audio/train/dad/4 4 2.wav
    july 1 2 5 /home/{user}/kaldi-trunk/egs/digits/digits audio/train/july/1 2 5.wav
    july 6 8 3
/home/{user}/kaldi-trunk/egs/digits/digits audio/train/july/6 8 3.wav
    #等等...
```

（3）text。该文件包含与其文本转录匹配的每个发言。

模式：<uterranceID> <text_transcription>

例如：

```
    dad_4_4_2 four four two
    july 1 2 5 one two five
    july 6 8 3 six eight three
    # 等等...
```

（4）utt2spk。该文件告诉 ASR 系统发言属于哪个特定的说话人。

模式：<uterranceID> <speakerID>

例如：

```
dad 4 4 2 dad
july 1 2 5 july
july 6 8 3 july
#等等...
```

（5）corpus.txt。该文件目录略有不同。在 kaldi-trunk/egs/digits/data 中创建另一个文件夹 local。在 kaldi-trunk/egs/digits/data/local 中创建一个文件 corpus.txt。该文件应该包含可能发生在 ASR 系统中的每一个话语转录（在我们的情况下，是 100 行 100 个音频文件）。

模式：<text_transcription>

例如：

```
one two five
six eight three
four four two
#等等...
语言数据
```

本节涉及语言建模文件，也需要将其视为“必须完成”。在这里查找语法细节：数据准备（精确描述每个文件）。还可以自己阅读其他例子脚本中的一些例子。现在展示的是一个理想的例子。

在 kaldi-trunk/egs/digits/data/local 目录中，创建一个文件夹 dict。在 kaldi-trunk/egs/digits/data/local/dict 中创建以下文件：

（1）lexicon.txt。该文件包含字典中的每个单词及其电话录音（取自/egs/voxforge）。

模式：<word> <phone 1> <phone 2> ...

例如：

```
!SIL sil
<UNK> spn
eight ey t
five f ay v
four f ao r
nine n ay n
one hh w ah n
one w ah n
seven s eh v ah n
six s ih k s
three th r iy
two t uw
zero z ih r ow
zero z iy r ow
```

（2）nonsilence_phones.txt。该文件列出了项目中存在的非静止的音素。

模式：<phone>

```
ah
```

```
ao
ay
eh
ey
f
hh
ih
iy
k
n
ow
r
s
t
th
uw
w
v
z
```

（3）silence_phones.txt。该文件列出了静止的音素。

模式：<phone>

```
sil
spn
```

（4）optional_silence.txt。该文件列出了可选的静止音素。

模式：<phone>

```
sil
```

最后，添加在示例性脚本中广泛使用的必需的 Kaldi 工具。

从 kaldi-trunk/egs/wsj/s5 中复制 utils 和 steps 两个文件夹及其全部内容，并将它们放在 kaldi-trunk/egs/digits 目录中。还可以创建到这些目录的链接，并可以在例如 kaldi-trunk/egs/voxforge/s5 中找到这样的链接。

打分脚本有助于获得解码结果。

从 kaldi-trunk/egs/voxforge/s5/local 中将脚本 score.sh 复制到项目中的相似位置（kaldi-trunk/egs/digits/local）。

还需要安装在这个示例中使用的语言建模工具包——SRI 语言建模工具包（SRILM）。有关详细的安装说明，请访问 kaldi-trunk/tools/install_srilm.sh（阅读里面所有的注释）。因为 SRILM 是一个商用需要收费的软件，所以没有自动下载的脚本。

下载 srilm-1.7.2.tar.gz 并重命名为 srilm.tgz，然后运行 install_srilm.sh。

```
mv ./srilm-1.7.2.tar.gz  ./srilm.tgz
./install_srilm.sh
```

在 kaldi-trunk/egs/digits 中创建一个文件夹 conf。在 kaldi-trunk/egs/digits/conf 内部创建两个文件（对于解码和 mfcc 特征提取过程中的一些配置修改，取自/egs/voxforge）：

（1）decode.config。

```
first_beam=10.0
beam=13.0
lattice_beam=6.0
```

（2）mfcc.conf。

```
--use-energy=false
```

最后一项工作是准备运行脚本来创建 ASR 系统。

- MONO：单声道训练。
- TRI1：简单的三音素训练。

这两种方法足以在仅使用数字词典和小训练数据集的情况下显示解码结果的显著差异。

在 kaldi-trunk/egs/digits 目录中创建 3 个脚本：

（1）cmd.sh。

```
#设置本地系统任务(本地 CPU – 无需外部群集)
export train_cmd=run.pl
export decode_cmd=run.pl
```

（2）path.sh。

```
#定义 Kaldi 根目录
export KALDI_ROOT='pwd'/../..

#设置路径以包含有用的工具
export PATH=$PWD/utils/:$KALDI_ROOT/src/bin:$KALDI_ROOT/tools/openfst/bin:
$KALDI_ROOT/src/fstbin/:$KALDI_ROOT/src/gmmbin/:$KALDI_ROOT/src/featbin/:$KAL
DI_ROOT/src/lmbin/:$KALDI_ROOT/src/sgmm2bin/:$KALDI_ROOT/src/fgmmbin/:$KALDI_
ROOT/src/latbin/:$PWD:$PATH

#定义音频数据路径
export DATA_ROOT="/home/{user}/kaldi-trunk/egs/digits/digits_audio"

#启用 SRILM
source $KALDI_ROOT/tools/env.sh

#为了数据能够正确排序所需的变量
export LC_ALL=C
```

（3）run.sh。

```
#!/bin/bash

. ./path.sh || exit 1
. ./cmd.sh || exit 1

nj=1          #并行任务的数量--对于这样一个小数据集 1 就很好
lm_order=1    #语言模型阶数（n-gram 数量）——对于数字文法来说，1 是足够的
```

```
    #安全机制(可能使用修改后的参数运行此脚本)
    . utils/parse options.sh || exit 1
    [[ $# -ge 1 ]] && { echo "Wrong arguments!"; exit 1; }

    #删除以前创建的数据(从上次的 run.sh 执行)
    rm -rf exp mfcc data/train/spk2utt data/train/cmvn.scp data/train/feats.scp
data/train/split1 data/test/spk2utt data/test/cmvn.scp data/test/feats.scp data/
test/split1 data/local/lang data/lang data/local/tmp data/local/dict/lexiconp.
txt

    echo
    echo "===== PREPARING ACOUSTIC DATA ====="
    echo

    #需要手工准备(或使用自己写的脚本):
    #
    # spk2gender  [<speaker-id> <gender>]
    # wav.scp     [<uterranceID> <full path to audio file>]
    # text          [<uterranceID> <text transcription>]
    # utt2spk     [<uterranceID> <speakerID>]
    # corpus.txt  [<text transcription>]

    #制作 spk2utt 文件
    utils/utt2spk to spk2utt.pl data/train/utt2spk > data/train/spk2utt
    utils/utt2spk_to_spk2utt.pl data/test/utt2spk > data/test/spk2utt

    echo
    echo "===== FEATURES EXTRACTION ====="
    echo

    # 制作 feats.scp 文件
    mfccdir=mfcc
    #如果在数据排序方面遇到任何问题,就在下面的脚本中取消注释并修改参数
    # utils/validate_data_dir.sh data/train  #检查准备好的数据的脚本,在这里路径是
data/train
    # utils/fix_data_dir.sh data/train       #数据正确排序的工具,在这里路径是
data/train
    steps/make mfcc.sh --nj $nj --cmd "$train cmd" data/train exp/make mfcc/train
$mfccdir
    steps/make mfcc.sh --nj $nj --cmd "$train cmd" data/test exp/make mfcc/test
$mfccdir

    # 制作 cmvn.scp 文件
    steps/compute cmvn stats.sh data/train exp/make mfcc/train $mfccdir
    steps/compute cmvn stats.sh data/test exp/make mfcc/test $mfccdir

    echo
    echo "===== PREPARING LANGUAGE DATA ====="
```

```
    echo

    #需要手工准备(或使用自己写的脚本)
    #
    # lexicon.txt                [<word> <phone 1> <phone 2> ...]
    # nonsilence phones.txt      [<phone>]
    # silence_phones.txt         [<phone>]
    # optional silence.txt       [<phone>]

    #准备语言数据
    utils/prepare lang.sh data/local/dict "<UNK>" data/local/lang data/lang

    echo
    echo "===== LANGUAGE MODEL CREATION ====="
    echo "===== MAKING lm.arpa ====="
    echo

    loc='which ngram-count';
    if [ -z $loc ]; then
      if uname -a | grep 64 >/dev/null; then
              sdir=$KALDI ROOT/tools/srilm/bin/i686-m64
      else
                      sdir=$KALDI ROOT/tools/srilm/bin/i686
      fi
      if [ -f $sdir/ngram-count ]; then
                      echo "Using SRILM language modelling tool from $sdir"
                      export PATH=$PATH:$sdir
      else
                      echo "SRILM toolkit is probably not installed.
                             Instructions: tools/install srilm.sh"
                      exit 1
      fi
    fi

    local=data/local
    mkdir $local/tmp
    ngram-count -order $lm order -write-vocab $local/tmp/vocab-full.txt -wbdiscount
-text $local/corpus.txt -lm $local/tmp/lm.arpa

    echo
    echo "===== MAKING G.fst ====="
    echo

    lang=data/lang
    arpa2fst --disambig-symbol=#0 --read-symbol-table=$lang/words.txt $local/
tmp/lm.arpa $lang/G.fst

    echo
    echo "===== MONO TRAINING ====="
```

```
    echo

    steps/train mono.sh --nj $nj --cmd "$train cmd" data/train data/lang exp/mono
|| exit 1

    echo
    echo "===== MONO DECODING ====="
    echo

    utils/mkgraph.sh --mono data/lang exp/mono exp/mono/graph || exit 1
    steps/decode.sh --config conf/decode.config --nj $nj --cmd "$decode_cmd"
exp/mono/graph data/test exp/mono/decode

    echo
    echo "===== MONO ALIGNMENT ====="
    echo

    steps/align si.sh --nj $nj --cmd "$train cmd" data/train data/lang exp/mono
exp/mono ali || exit 1

    echo
    echo "===== TRI1 (first triphone pass) TRAINING ====="
    echo

    steps/train deltas.sh --cmd "$train cmd" 2000 11000 data/train data/lang
exp/mono ali exp/tri1 || exit 1

    echo
    echo "===== TRI1 (first triphone pass) DECODING ====="
    echo

    utils/mkgraph.sh data/lang exp/tri1 exp/tri1/graph || exit 1
    steps/decode.sh --config conf/decode.config --nj $nj --cmd "$decode cmd"
exp/tri1/graph data/test exp/tri1/decode

    echo
    echo "===== run.sh script is finished ====="
    echo
```

现在要做的就是运行 run.sh 脚本。终端的日志能提示你如何处理可能遇到的错误。

除了在终端窗口中会注意到某些解码结果外，还要进入新制作的 kaldi-trunk/egs/digits/exp。可能会注意到，文件夹有同样目录结构的 mono 和 tri1 结果。来到 mono/decode 目录，可能会找到结果文件（以 wer_ {number}方式命名）。解码过程的日志可以在日志文件夹（同一目录）中找到。

在成功安装 Kaldi 后，可运行一些示例脚本（Yesno、Voxforge、LibriSpeech，它们相对容易，有免费的声学/语言数据供下载）。

3.3 使用 FFmpeg 提取音频

可以使用 Ffmpeg 开发包转换各种音频格式或者从视频中提取音频。

在 CentOS 7 下，首先安装 YUM 源：

```
    sudo rpm --import http://li.nux.ro/download/nux/RPM-GPG-KEY-nux.ro
    sudo rpm -Uvh http://li.nux.ro/download/nux/dextop/el7/x86 64/nux-dextop-
release-0-5.el7.nux.noarch.rpm
```

然后安装 FFmpeg 和 FFmpeg 开发包：

```
sudo yum install ffmpeg ffmpeg-devel -y
```

测试是否安装成功：

```
#ffmpeg
```

如果想了解更多关于 FFmpeg 的命令行参数，可以输入：

```
#ffmpeg -h
```

使用 FFmpeg 将 WAV 格式转为 MP3 格式：

```
# ffmpeg -i A.wav -acodec libmp3lame A.mp3
```

将 OGG 格式转为 MP3 格式：

```
# ffmpeg -i audio.ogg -acodec libmp3lame audio.mp3
```

将 AC3 格式转为 MP3 格式：

```
#ffmpeg -i audio.ac3 -acodec libmp3lame audio.mp3
```

将 AAC 格式转为 MP3 格式：

```
#ffmpeg -i audio.aac -acodec libmp3lame audio.mp3
```

3.4 时间序列

可以用一些时间序列分析方法来分析语音。TimeSeriesPoint 类表示支持矢量的时间序列点：

```
public class TimeSeriesPoint {
    private final double[] measurements;  //矢量值
    private final int hashCode;

    public TimeSeriesPoint(double[] values) {
        int hashCode = 0;
        measurements = new double[values.length];
        for (int i = 0; i < values.length; i++) {
```

```
            hashCode += new Double(values[i]).hashCode();
            measurements[i] = values[i];
        }
        this.hashCode = hashCode;
    }

    public double get(int dimension) {
        return measurements[dimension];
    }

    public double[] toArray() {
        return measurements;
    }

    public int size() {
        return measurements.length;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        else if (o instanceof TimeSeriesPoint) {
            final double[] testValues = ((TimeSeriesPoint) o).toArray();
            if (testValues.length == measurements.length) {
                for (int x = 0; x < measurements.length; x++)
                    if (measurements[x] != testValues[x])
                        return false;

                return true;
            } else
                return false;
        } else
            return false;
    }

    @Override
    public int hashCode() {
        return hashCode;
    }

}
```

TimeSeriesItem 类表示时间序列项：

```
public final class TimeSeriesItem {
    private final double time;
    private final TimeSeriesPoint point;

    public TimeSeriesItem(double time, TimeSeriesPoint point) {
        this.time = time;
```

```
            this.point = point;
        }

        public double getTime() {
            return time;
        }

        public TimeSeriesPoint getPoint() {
            return point;
        }

    }
```

3.5 动态时间规整

对于时间序列，使用最经常使用的欧几里得距离来计算相似度存在很明显的缺陷。举个比较简单的例子，序列 A：1,1,1,10,2,3，序列 B：1,1,1,2,10,3，如果用欧几里得距离，也就是 distance[i][j]=(b[j]-a[i])*(b[j]-a[i])来计算，则总的距离和是 128，应该说这个距离是非常大的，而实际上这个序列的图像是十分相似的。这种情况下就有人开始考虑寻找新的时间序列距离的计算方法，然后提出了 DTW(dynamic time warping，动态时间规整)算法。这种算法在语音识别、机器学习方面有着很重要的作用。

这个算法是基于动态规划的思想，解决了发音长短不一的模板匹配问题，简单来说，就是通过构建一个邻接矩阵，寻找最短路径和。

还以上面的两个序列作为例子，A 中的 10 和 B 中的 2 对应以及 A 中的 2 和 B 中的 10 对应的时候，distance[3]以及 distance[4]肯定是非常大的，这就直接导致了最后距离和的膨胀，这种时候，需要调整一下时间序列，如果让 A 中的 10 和 B 中的 10 对应，A 中的 1 和 B 中的 2 对应，那么最后的距离和就将大大缩短，这种方式可以被看作一种时间扭曲。看到这里，相信应该会有人提出来，为什么不能使用 A 中的 2 与 B 中的 2 对应的问题，那样的话距离和肯定是 0 了啊，距离应该是最小的吧，但这种情况是不允许的，因为 A 中的 10 是发生在 2 的前面，而 B 中的 2 则发生在 10 的前面，如果对应方式交叉，则会导致时间上的混乱，不符合因果关系。两个序列对齐的方式如图 3-2 所示。

接下来，以 output[5][5]（所有的记录下标从 0 开始，开始时全部置 0）记录 A 和 B 之间的 DTW 距离。简单介绍一下具体的算法：这个算法其实就是一个如图 3-3 所示的动态规划，其循环等式是：

```
    output[i][j]=Min(Min(output[i-1][j],output[i][j-1]),output[i-1][j-1])+dis
tance[i][j];
```

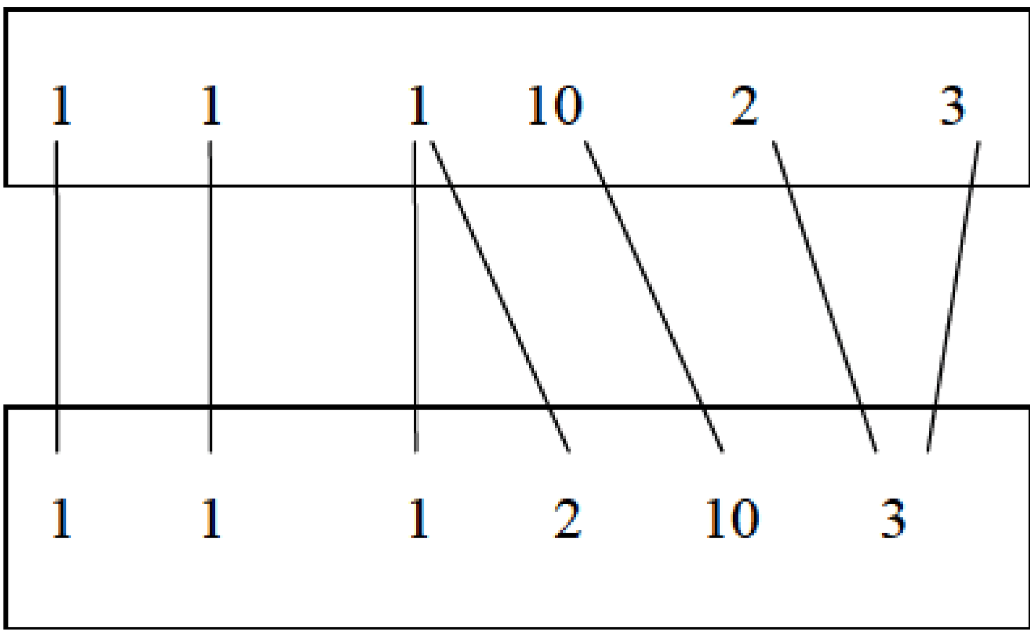


图 3-2　两个序列对齐

最后得到的 output[5][5]就是所需要的 DTW 距离。

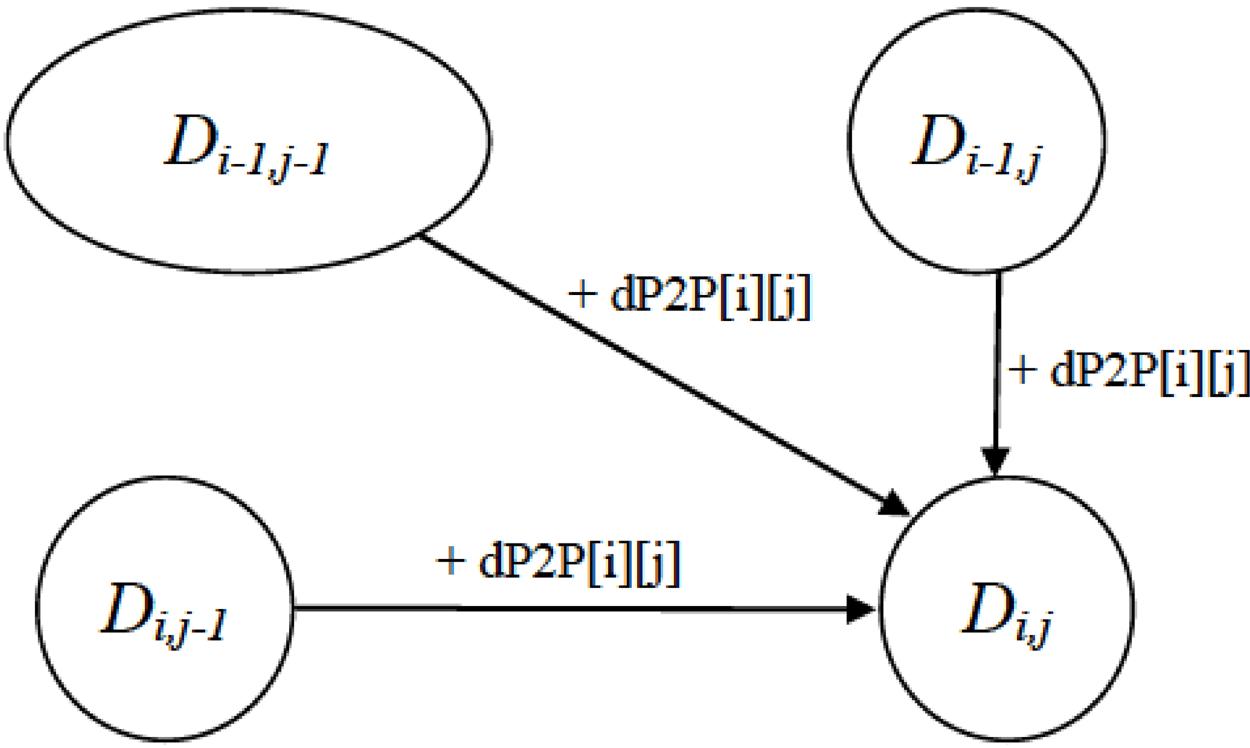


图 3-3　DTW 计算依赖关系图

DTW 距离实现代码如下：

```
//计算两个点的相似度
private static double pointDistance(int i, int j, double[] ts1, double[] ts2) {
      double diff = ts1[i] - ts2[j];
      return (diff * diff);
}

//计算两个序列 ts1 和 ts2 的距离
public static double DTWDistance(double[] ts1, double[] ts2) {
    int i, j;

    //构建一个点对点的距离矩阵
    double[][] dP2P = new double[ts1.length][ts2.length];
    for (i = 0; i < ts1.length; i++) {
        for (j = 0; j < ts2.length; j++) {
            dP2P[i][j] = pointDistance(i, j, ts1, ts2);
        }
    }

    //用动态规划算法构建最优距离矩阵
    double[][] D = new double[ts1.length][ts2.length];
```

```
        D[0][0] = dP2P[0][0]; //开始点

        for (i = 1; i < ts1.length; i++) { //用最优值填充距离矩阵的第一列
            D[i][0] = dP2P[i][0] + D[i - 1][0];
        }

        for (j = 1; j < ts2.length; j++) { //用最优值填充距离矩阵的第一行
            D[0][j] = dP2P[0][j] + D[0][j - 1];
        }

        for (i = 1; i < ts1.length; i++) { //填充剩下的
            for (j = 1; j < ts2.length; j++) {
                double[] steps = { D[i - 1][j - 1], D[i - 1][j], D[i][j - 1] };
                double min = Math.min(steps[0], Math.min(steps[1], steps[2]));
                D[i][j] = dP2P[i][j] + min;
            }
        }

        i = ts1.length - 1;
        j = ts2.length - 1;
        return D[i][j];
    }
```

测试：

```
    public static void main(String[] args) {
        double[] ts1 = { 1,1,1,10,2,3};             //第一个序列
        double[] ts2 = { 1,1,1,2,10,3};             //第二个序列

        System.out.println(DTWDistance(ts1,ts2));   //输出两个序列的相似度
    }
```

输出 2。因为序列 ts1 中的 1 对应序列 ts2 中的 2，序列 ts1 中的 2 对应序列 ts2 中的 3，这两点的差加起来是 2。

3.6 傅里叶变换

将音频数据提取到一个数组后，可以将 N 个数据样本传递给计算离散傅里叶变换的函数（或更有效的快速傅里叶变换）。

3.6.1 离散傅里叶变换

离散傅里叶变换（DFT）是数字信号处理（DSP）的一种基本但非常通用的算法。

这里逐步介绍从头开始实现算法的步骤。

音频信号可以分解成不同振幅、频率、相位的正弦波的叠加。波的相位用复数来表示。

DFT 总体上是一个将 n 个复数的向量映射到 n 个复数的另一个向量的函数。使用基于 0 的索引，令 $\boldsymbol{x}(t)$表示输入向量的第 t 个元素，并让 $\boldsymbol{X}(k)$表示输出向量的第 k 个元素。然后基本的 DFT 由以下公式给出：

$$\boldsymbol{X}(k)=\sum_{t=0}^{n-1}\boldsymbol{x}(t)\mathrm{e}^{-2\pi itk/n}$$

式中，向量 $\boldsymbol{x}$ 表示各个时间点的信号水平；向量 $\boldsymbol{X}$ 表示各个频率下的信号水平；公式表示频率 k 处的信号水平等于{每个时间 t 的信号水平乘以复数指数}的和。

DFT 采用 n 个复数的输入向量，并计算 n 个复数的输出向量。由于 Java 没有原生的复数类型，所以我们将用一对实数手动模拟一个复数。向量是一个数字序列，可以用数组表示。我们不会返回输出数组，而是通过引用将它们作为参数传入。我们来写一个骨架方法：

```
void dft(double[] inreal , double[] inimag,
        double[] outreal, double[] outimag) {
    //假设所有 4 个数组具有相同的长度
    int n = inreal.length;
    //未完成的方法体
}
```

接下来，编写外部循环为每个输出元素分配一个值：

```
void dft(double[] inreal , double[] inimag,
        double[] outreal, double[] outimag) {
    int n = inreal.length;
    for (int k = 0; k < n; k++) {  //对于每个输出元素
        outreal[k] = ?;  //未完成
        outimag[k] = ?;  //未完成
    }
}
```

求和符号虽然看起来很吓人，但实际上很容易理解。有限求和的一般形式仅仅意味着：

$$\sum_{j=a}^{b}f(j)=f(a)+f(a+1)+\cdots+f(b-1)+f(b)$$

看看我们如何取代 j 的值？在代码中，它看起来像这样：

```
double sum = 0;
for (int j = a; j <= b; j++) {
    sum += f(j);
}
```

```
//sum 的值现在有了想要的答案
```

复数的算术：

复数的加法很容易：

```
(a+bi) + (c+di) = (a+c) + (b+d)i
```

复数的乘法稍微困难一些，使用分配律和 $i^2 = -1$：

$$(a+bi)(c+di) = ac + adi + bci - bd = (ac-bd) + (ad+bc)i$$

欧拉公式告诉我们，对于任何实数 x 有：$e^{xi} = \cos x + i\sin x$。

而且，余弦是偶函数。因此 $\cos(-x) = \cos x$。正弦是奇函数，所以 $\sin(-x) = -(\sin x)$。

通过替换：

$$e^{-2\pi i tk/n} = e^{(-2\pi tk/n)i} = \cos\left(-2\pi\frac{tk}{n}\right) + i\sin\left(-2\pi\frac{tk}{n}\right) = \cos\left(2\pi\frac{tk}{n}\right) - i\sin\left(2\pi\frac{tk}{n}\right)$$

设 $\mathrm{Re}(x)$是 x 的实部，并设 $\mathrm{Im}(x)$是 x 的虚部。根据定义，$x = \mathrm{Re}(x) + i\mathrm{Im}(x)$。因此：

$$x(t)e^{-2\pi i tk/n} = [\mathrm{Re}(x(t)) + i\mathrm{Im}(x(t))]\left[\cos\left(2\pi\frac{tk}{n}\right) - i\sin\left(2\pi\frac{tk}{n}\right)\right]$$

展开复数乘法得到：

$$x(t)e^{-2\pi i tk/n} = \left[\mathrm{Re}(x(t))\cos\left(2\pi\frac{tk}{n}\right) + \mathrm{Im}(x(t))\sin\left(2\pi\frac{tk}{n}\right)\right]$$
$$+ i\left[-\mathrm{Re}(x(t))\sin\left(2\pi\frac{tk}{n}\right) + \mathrm{Im}(x(t))\cos\left(2\pi\frac{tk}{n}\right)\right]$$

因此，总和中的每一项都有这个实部和虚部的代码：

```
double angle = 2 * Math.PI * t * k / n;
double real =  inreal[t] * Math.cos(angle) + inimag[t] * Math.sin(angle);
double imag = -inreal[t] * Math.sin(angle) + inimag[t] * Math.cos(angle);
```

将每个项目求和的代码合并到总的代码中，我们就完成了：

```
static void dft(double[] inreal , double[] inimag,
                double[] outreal, double[] outimag) {
    int n = inreal.length;
    for (int k = 0; k < n; k++) {  //对于每个输出元素
        double sumreal = 0;
        double sumimag = 0;
        for (int t = 0; t < n; t++) {  //对于每个输入元素
            double angle = 2 * Math.PI * t * k / n;
            sumreal +=  inreal[t] * Math.cos(angle) + inimag[t] * Math.sin(angle);
            sumimag += -inreal[t] * Math.sin(angle) + inimag[t] * Math.cos(angle);
        }
        outreal[k] = sumreal;
```

```
            outimag[k] = sumimag;
        }
    }
```

3.6.2　快速傅里叶变换

有多种离散傅里叶变换（DFT）的快速算法，最常见的是 Cooley-Tukey 算法。目前一般所指的快速傅里叶变换（FFT）算法就是 Cooley-Tukey 算法。这一方法以分治法为策略递归地将长度为 $N=N_1N_2$ 的离散傅里叶变换分解为长度为 N_1 的 N_2 个较短序列的离散傅里叶变换，以及与 $O(N)$个旋转因子的复数乘法。

最简单的情况：假设 N=2。

$$\boldsymbol{X}(k)=\sum_{t=0}^{n-1}\boldsymbol{x}(t)\mathrm{e}^{-2\pi\,\mathrm{i}tk/n}=\boldsymbol{x}(0)+\boldsymbol{x}(1)\mathrm{e}^{-\pi\,\mathrm{i}k}$$

当 k=0 时，$\boldsymbol{X}(0)=\boldsymbol{x}(0)+\boldsymbol{x}(1)$

当 k=1 时，$\boldsymbol{X}(1)=\boldsymbol{x}(0)-\boldsymbol{x}(1)$

计算给定复向量的 DFT，并将结果存回到向量中。向量的长度必须是 2 的乘方。使用 Cooley-Tukey 算法（按时间抽取基数 2 算法）。

```
    public static void transformRadix2(double[] real, double[] imag) {
        //长度变量
        int n = real.length;
        if (n != imag.length)
          throw new IllegalArgumentException("Mismatched lengths");
        int levels = 31 - Integer.numberOfLeadingZeros(n);
          //Equal to floor(log2(n))
        if (1 << levels != n)
          throw new IllegalArgumentException("Length is not a power of 2");

        //构建三角函数表
        double[] cosTable = new double[n / 2];
        double[] sinTable = new double[n / 2];
        for (int i = 0; i < n / 2; i++) {
          cosTable[i] = Math.cos(2 * Math.PI * i / n);
          sinTable[i] = Math.sin(2 * Math.PI * i / n);
        }

        //位反转寻址置换
        for (int i = 0; i < n; i++) {
          int j = Integer.reverse(i) >>> (32 - levels);
          if (j > i) {
              double temp = real[i];
              real[i] = real[j];
              real[j] = temp;
```

```
                temp = imag[i];
                imag[i] = imag[j];
                imag[j] = temp;
            }
        }

        //Cooley-Tukey 以 2 为基数按时间抽取的 FFT
        for (int size = 2; size <= n; size *= 2) {
            int halfsize = size / 2;
            int tablestep = n / size;
            for (int i = 0; i < n; i += size) {
                for (int j = i, k = 0; j < i + halfsize; j++, k += tablestep) {
                    int l = j + halfsize;
                    double tpre =  real[l] * cosTable[k] + imag[l] * sinTable[k];
                    double tpim = -real[l] * sinTable[k] + imag[l] * cosTable[k];
                    real[l] = real[j] - tpre;
                    imag[l] = imag[j] - tpim;
                    real[j] += tpre;
                    imag[j] += tpim;
                }
            }
            if (size == n)  //防止溢出
                break;
        }
    }
```

计算任意长度向量的 FFT 需要使用循环卷积。

计算给定复数向量的循环卷积。每个向量的长度必须相同。

```
    public static void convolve(double[] xreal, double[] ximag,
            double[] yreal, double[] yimag, double[] outreal, double[] outimag) {

        int n = xreal.length;
        if (n != ximag.length || n != yreal.length || n != yimag.length
                || n != outreal.length || n != outimag.length)
            throw new IllegalArgumentException("Mismatched lengths");

        xreal = xreal.clone();
        ximag = ximag.clone();
        yreal = yreal.clone();
        yimag = yimag.clone();
        //计算给定复向量的 DFT，并将结果存回到向量中
        transform(xreal, ximag);
        transform(yreal, yimag);

        for (int i = 0; i < n; i++) {
            double temp = xreal[i] * yreal[i] - ximag[i] * yimag[i];
            ximag[i] = ximag[i] * yreal[i] + xreal[i] * yimag[i];
            xreal[i] = temp;
```

```
        }
        inverseTransform(xreal, ximag);     //逆变换

        for (int i = 0; i < n; i++) {       //缩放（因为这个 FFT 实现省略了它）
          outreal[i] = xreal[i] / n;
          outimag[i] = ximag[i] / n;
        }
}
```

不限制向量长度的 FFT。使用 Bluestein 线性调频 *Z* 变换算法实现：

```
public static void transformBluestein(double[] real, double[] imag) {
        //找出 2 的幂卷积长度 m，使得 m> = n * 2 + 1
        int n = real.length;
        if (n != imag.length)
          throw new IllegalArgumentException("Mismatched lengths");
        if (n >= 0x20000000)
          throw new IllegalArgumentException("Array too large");
        int m = Integer.highestOneBit(n) * 4;

        //构建三角函数表
        double[] cosTable = new double[n];
        double[] sinTable = new double[n];
        for (int i = 0; i < n; i++) {
          int j = (int)((long)i * i % (n * 2));  //这比 j = i * i 更准确
          cosTable[i] = Math.cos(Math.PI * j / n);
          sinTable[i] = Math.sin(Math.PI * j / n);
        }

        //临时向量和预处理
        double[] areal = new double[m];
        double[] aimag = new double[m];
        for (int i = 0; i < n; i++) {
          areal[i] =  real[i] * cosTable[i] + imag[i] * sinTable[i];
          aimag[i] = -real[i] * sinTable[i] + imag[i] * cosTable[i];
        }
        double[] breal = new double[m];
        double[] bimag = new double[m];
        breal[0] = cosTable[0];
        bimag[0] = sinTable[0];
        for (int i = 1; i < n; i++) {
          breal[i] = breal[m - i] = cosTable[i];
          bimag[i] = bimag[m - i] = sinTable[i];
        }

        //卷积
        double[] creal = new double[m];
        double[] cimag = new double[m];
        convolve(areal, aimag, breal, bimag, creal, cimag);
```

```
        //后处理
        for (int i = 0; i < n; i++) {
          real[i] =  creal[i] * cosTable[i] + cimag[i] * sinTable[i];
          imag[i] = -creal[i] * sinTable[i] + cimag[i] * cosTable[i];
        }
    }
```

计算给定复向量的 DFT，并将结果存回到向量中。向量可以有任何长度。这是一个包装函数。

```
    public static void transform(double[] real, double[] imag) {
        int n = real.length;
        if (n != imag.length)
          throw new IllegalArgumentException("Mismatched lengths");
        if (n == 0)
          return;
        else if ((n & (n - 1)) == 0)  //是 2 的幂
          transformRadix2(real, imag);
        else  //用于任意长度的更复杂的算法
          transformBluestein(real, imag);
    }
```

可以根据 FFT 的输出结果得到幅度和相位。幅度编码为复数的模。Audacity 中显示的是频谱图，其中 Y 轴的值可以被看作复数的模（sqrt(x^2+y^2)），而相位被编码为角度（atan2(y,x)）。

正频率代表了逆时针的圆周运动，负频率代表了顺时针的圆周运动。

3.7　MFCC 特征

以梅尔频率倒普系数（MFCC）为例，对于语音信号处理如下：

（1）输入 16kHz 采样音频。

（2）以一个 25ms 的窗口（每次移动 10ms 将输出一个矢量序列，即每 10ms 一个）产生数值序列。

（3）乘以窗口函数，例如海明距离。

（4）执行 FFT。

（5）在每个频率桶中记录能量。

（6）做离散余弦变换（DCT），得到“倒谱”。

（7）保留倒谱的前 13 个系数。

在深度学习中，可以直接读取声音的波形文件，不用 MFCC 特征。

3.8 在线解码

首先介绍使用现成的模型识别英语，然后介绍使用 Alex-ASR 实现在线解码。

3.8.1 使用现成的模型

使用已构建的 online-nnet2 模型的示例：

在本节中，将解释如何从 www.kaldi-asr.org 下载已构建的 online-nnet2 模型，并根据自己的数据对其进行评估。

在一个新创建的目录下使用以下命令下载一个识别英语的模型存档并将其解压缩：

```
    #wget  http://kaldi-asr.org/downloads/build/5/trunk/egs/fisher english/s5/
exp/nnet2 online/nnet a gpu online/archive.tar.gz -O nnet a gpu online.tar.gz
    #wget  http://kaldi-asr.org/downloads/build/2/sandbox/online/egs/fisher english/
s5/exp/tri5a/graph/archive.tar.gz -O graph.tar.gz
    #mkdir -p nnet a gpu online graph
    #tar zxvf nnet_a_gpu_online.tar.gz -C nnet_a_gpu_online
    #tar zxvf graph.tar.gz -C graph
```

这个模型位于 http://kaldi-asr.org/downloads/build/2/sandbox/online/egs/fisher_english/s5，是使用 fisher_english 配方构建的。

上面的命令下载了两个目录：exp/tri5a/graph 和 exp/nnet2_online/nnet_a_gpu_online。

修改配置文件中的路径名，可以用如下的 Shell 脚本实现：

```
    for x in nnet_a_gpu_online/conf/*conf; do
      cp $x $x.orig
      sed s:/export/a09/dpovey/kaldi-clean/egs/fisher english/s5/exp/nnet2 online/:
$(pwd)/: < $x.orig > $x
    done
```

可以把上面的代码写入 change_config.sh 文件中，然后执行：

```
    #sh change_config.sh
```

接下来，选择要解码的单个.wav 文件。可以通过如下命令来下载示例文件：

```
    #wget http://www.signalogic.com/melp/EngSamples/Orig/ENG_M.wav
```

这是在网上找到的 8kHz 采样的.wav 文件。可以使用以下命令对其进行解码：

```
    #/home/kaldi/src/online2bin/online2-wav-nnet2-latgen-faster --do-endpointing=
false \
        --online=false \
        --config=nnet a gpu online/conf/online nnet2 decoding.conf \
        --max-active=7000 --beam=15.0 --lattice-beam=6.0 \
        --acoustic-scale=0.1 --word-symbol-table=graph/words.txt \
       nnet a gpu online/final.mdl   graph/HCLG.fst   "ark:echo   utterance-id1
utterance-id1|" "scp:echo utterance-id1 ENG_M.wav|" \
       ark:/dev/null
```

输出结果如下：

```
    LOG(online2-wav-nnet2-latgen-faster[5.5.159~1420-46826]:ComputeDerivedVars():
ivector-extractor.cc:183) Computing derived variables for iVector extractor
    LOG  (online2-wav-nnet2-latgen-faster[5.5.159~1420-46826]:ComputeDerivedVars():
ivector-extractor.cc:204) Done.

    utterance-id1 tons of what on the way for races two miles and then in nineteen
ninety two by so let's say ooh [noise] three them all these to commemorate columbus
is drawn into the new world five hundred years ago i went to the moon is to promote
the use of so the sales in space exploration
    LOG (online2-wav-nnet2-latgen-faster[5.5.159~1420-46826]:main():online2-wav-
nnet2-latgen-faster.cc:276) Decoded utterance utterance-id1
    LOG (online2-wav-nnet2-latgen-faster[5.5.159~1420-46826]:Print():online-
timing.cc:55) Timing stats: real-time factor for offline decoding was 1.24083 =
20.4748
    seconds  / 16.5009 seconds.
    LOG (online2-wav-nnet2-latgen-faster[5.5.159~1420-46826]:main():online2-wav-
nnet2-latgen-faster.cc:282) Decoded 1 utterances, 0 with errors.
    LOG
(online2-wav-nnet2-latgen-faster[5.5.159~1420-46826]:main():online2-wav-nnet2
-latgen-faster.cc:284) Overall likelihood per frame was 0.285607 per frame over
 1648 frames.
```

为了识别中文，当前可以使用 CVTE 中文普通话识别模型（http://kaldi-asr.org/models/2/0002_cvte_chain_model.tar.gz），使用 Aishell(http://www.openslr.org/resources/33/data_aishell.tgz)中的.wav 文件 BAC009S0028W0121.wav 做测试。

将 0002_cvte_chain_model.tar.gz 解压到 kaldi/egs 下。

在/home/kaldi/src/online2bin 目录下运行：

```
    ./online2-wav-nnet3-latgen-faster --do-endpointing=false --online=false
--feature- type=fbank --fbank-config=../../egs/cvte/s5/conf/fbank.conf --max-
active=7000 --beam=15.0 --lattice-beam=6.0 --acoustic-scale=1.0 --word-symbol-
table=../../egs/cvte/s5/exp/chain/tdnn/graph/words.txt  ../../egs/cvte/s5/exp/
chain/tdnn/final.mdl ../../egs/cvte/s5/exp/chain/tdnn/graph/HCLG.fst 'ark:echo
utter1 utter1|' 'scp:echo utter1 /home/corpus/BAC009S0028W0121.wav|' ark:/dev/
null
```

3.8.2 使用 Alex-ASR

下载源代码：

```
#git clone https://github.com/choko/alex-asr.git
```

更新源：

```
#apt-get update
```

安装依赖包：

```
#apt-get install -y build-essential libatlas-base-dev python-dev python-pip
```

```
git wget gfortran g++ unzip zlib1g-dev automake autoconf libtool subversion
```

使用 pip 命令安装 Python 模块 cython：

```
#python -m pip install cython
```

安装 Alex-ASR：

```
#python setup.py install
```

运行识别数字的测试脚本：

```
#cd test/
#python ./test.py
```

示例用法：

```
from alex_asr import Decoder
import wave
import struct
import os

#从"asr_model_dir"目录加载语音识别模型
decoder = Decoder("asr model dir/")

#从输入.wav 文件加载音频帧
data = wave.open("input.wav")
frames = data.readframes(data.genframes())

#将音频数据馈送到解码器
decoder.accept_audio(frames)
decoder.decode(data.genframes())
decoder.input finished()

#获取并打印最佳假设
prob, word ids = decoder.get best path()
print " ".join(map(decoder.get_word, word_ids))
```

3.9　加权有限状态转换

因为大多数组件（语言模型、词典、词图）都可以用有限状态自动机描述，可以通过组合操作将不同的模型集成到一个模型中，所以采用加权有限状态转换（WFST）。WFST 是用于描述模型的统一框架。

级联多个 WFST。

H：HMM

C：上下文相关模型

L：词典

G：文法

级联 4 个 WFST 形式化的写法是：

H ○ C ○ L ○ G。

只需要将这个识别网络（WFST 网络）读入内存，然后基于声学模型就可以在这个网络上完成解码，不需要像原有系统那样同时考虑声学模型、词典、语言模型等。这样简化了语音识别系统的设计与实现。

3.9.1 FSA

有限状态接收器（finite state acceptor，FSA）接收一个字符串的集合。一个字符串是一个符号序列。对于给定的字符串，FSA 返回“接收”或者“不接收”两种结果。

将 FSA 视为可能是无限数量的字符串集的表示。

图 3-4 是一个只接收字符串 yes 和 no 的 FSA，即 set{yes,no}。

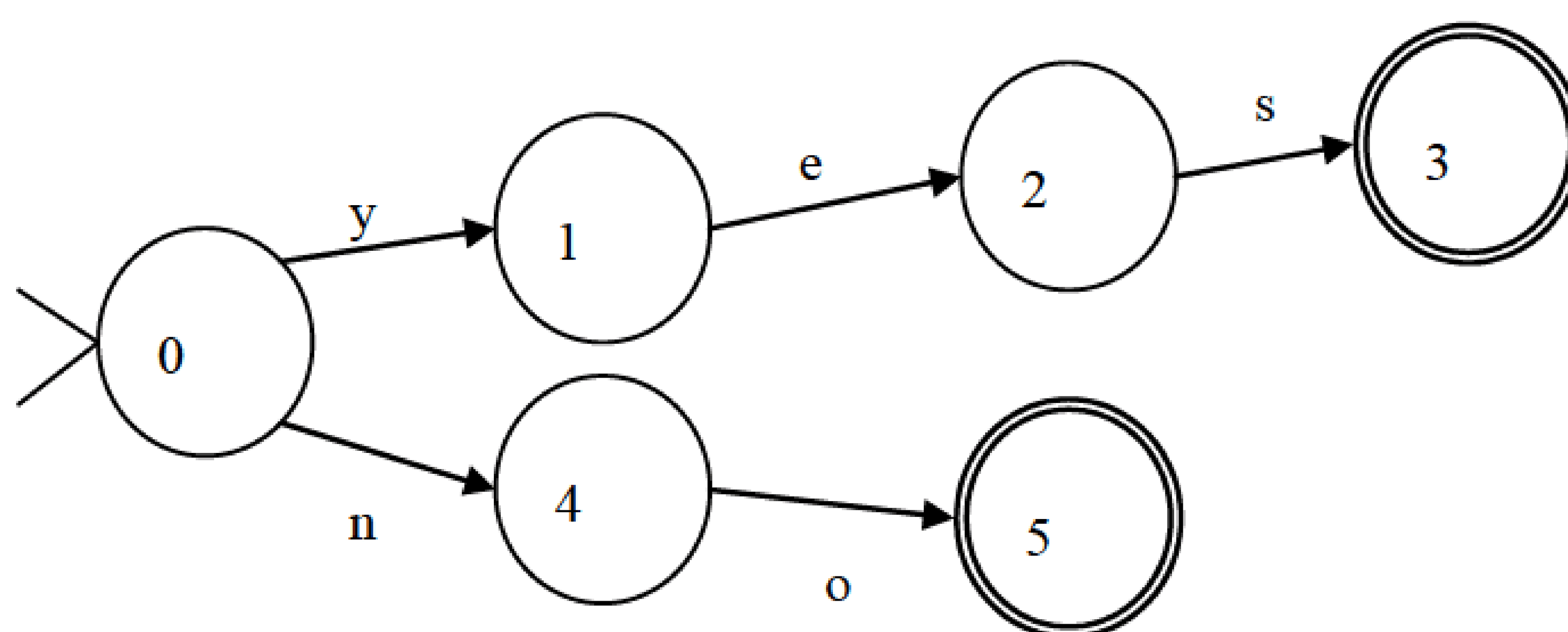


图 3-4 表示 yesno 的有限状态接收器

圈中的数字是状态标签（不是很重要）。

标签是附属于边上的符号。

箭头指向的节点是开始节点。双圈节点表示可以作为结束节点。开始节点只能有一个，而结束节点可以有多个。这样的图称为状态转换图。

dk.brics.automaton 是一个 FSA 的实现。使用它定义表示 yesno 的 FSA 代码如下：

```
String s1 = "yes";
String s2 = "no";
Automaton a= Automaton.makeString(s1);        //y e s 有限状态接收器
Automaton b= Automaton.makeString(s2);        //n o 有限状态接收器
Automaton c = BasicOperations.union(a, b);    //并运算

String word = "yes";
System.out.println(c.run(word));
```

3.9.2　FST

有限状态转换器（finite state transducer，FST）就是利用有限状态机把输入串映射成输出串。

例如判断二进制串的奇偶性，用两个状态 S1 和 S2 分别表示偶数和奇数。

使用图 3-5 所示的状态转换图来判断字符串中 1 的数量是奇数还是偶数，其状态转换表见表 3-1。例如：

1 0 1 1 0 0 1 → S1

0 0 0 1 0 0 0 → S2

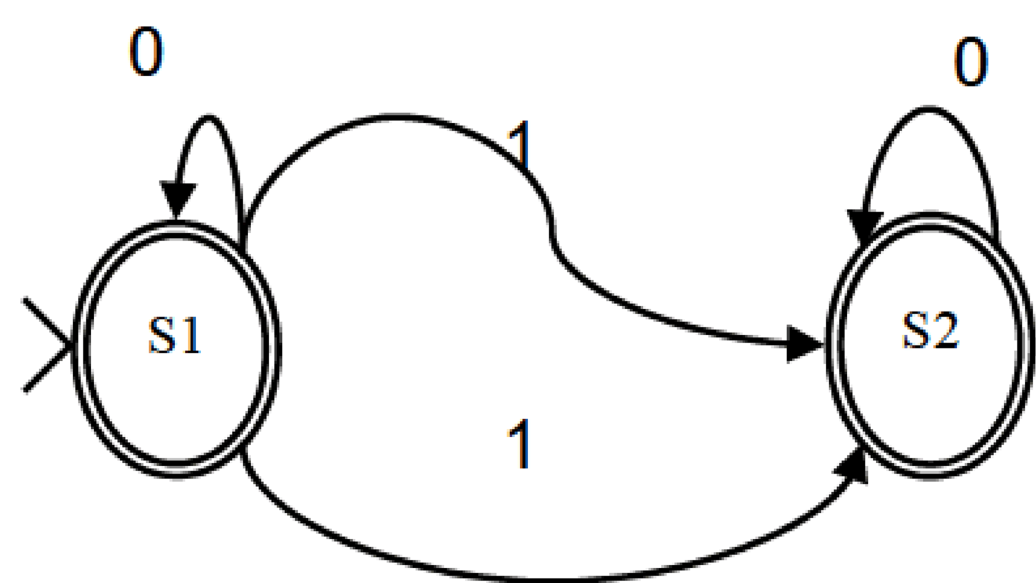


图 3-5　状态转换图

表 3-1　状态转换表

状　　态	转　　换
S1	0 → S1, 1 → S2
S2	0 → S2, 1 → S1

实现这个有限状态转换器的代码如下：

```
int parity(String s) {
      int state = 1;
      for(int i = 0;i< s.length();++i){
        char ch = s.charAt(i);
        switch (state){
            case 1:
                if(ch=='1')
                    state = 2;
                break;
            case 2:
                if(ch=='1')
                    state = 1;
                break;
        }
      }
      return state;
}
```

测试这个方法：

```
System.out.print(parity("01010")); //输出 1
```

为了构建最小完美散列，需要把排好序的单词{clear,clever,ear,ever,fat,father}映射到序号（0,1,2,…）。当遍历边的时候，把经过的值加起来，例如 father 在 f 命中 4 并且在 h 命中 1，因此它的输出是 5，如图 3-6 所示。

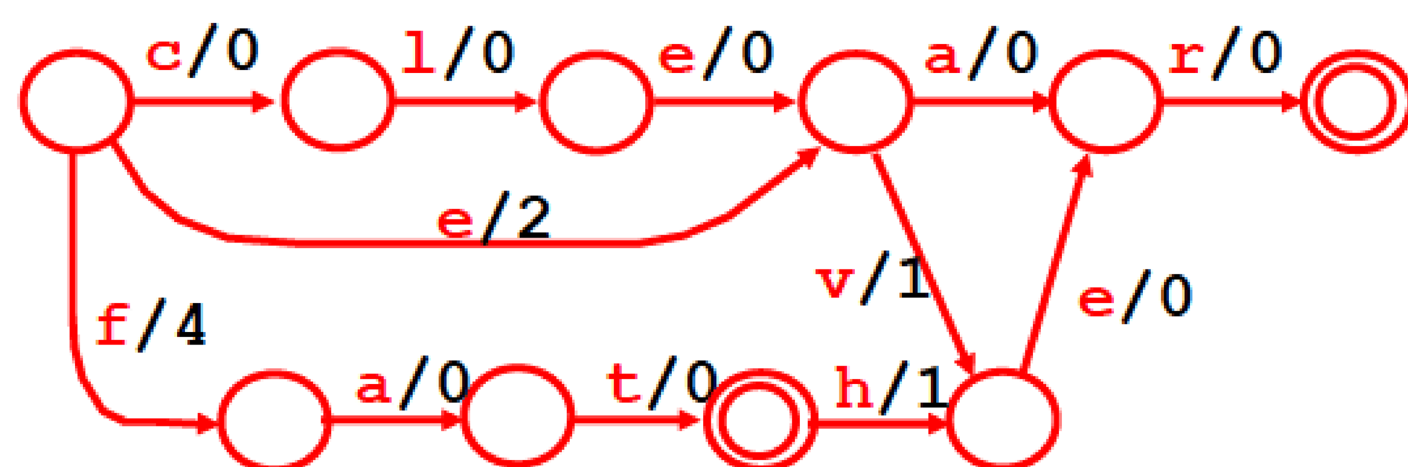


图 3-6　有限状态转换器

3.9.3　WFST

环是抽象代数中使用的基本代数结构之一。它由一个含有两个二元运算的集合组成，它可以推广加法运算和乘法运算。半环是与环相似的代数结构，但不要求每个元素必须具有加法可逆。

求和：计算序列的权重（用该序列标记的路径的权重之和）。

乘积：计算路径的权重（构成转换权重的乘积）。

常见的几类半环的比较见表 3-2。

表 3-2　几类半环的比较

Name	Set	⊕（求和）	⊗（乘积）	$\bar{0}$（零）	$\bar{1}$（一）
Boolean	{0,1}	∨	∧	0	1
Real	[0,∞]	+	*	0	1
Log	[-∞,∞]	-log　(exp(-a) + exp(-b))	+	∞	0
Tropical	[-∞,∞]	min	+	∞	0

所有实际应用都使用热带半环（Tropical Semiring），其最明显的实例是在语音识别系统中的假设中组合词语的负对数概率。

在 jopenfst(https://github.com/steveash/jopenfst)中使用热带半环的例子：

```
MutableFst fst = new MutableFst(TropicalSemiring.INSTANCE);
//用符号识别状态
fst.useStateSymbols();
MutableState startState = fst.newStartState("<start>");
//设定最终权重让这个状态成为合格的最终状态
fst.newState("</s>").setFinalWeight(0.0);
```

```
//可以将符号手动添加到符号表
int symbolId = fst.getInputSymbols().getOrAdd("<eps>");
fst.getOutputSymbols().getOrAdd("<eps>");

//直接加边
fst.addArc("state1", "inA", "outA", "state2", 1.0);

//也可以使用状态实例
fst.addArc(startState, "inC", "outD", fst.getOrNewState("state3"), 123.0);
```

3.10　语音识别语料库

本节介绍几个常用的语音识别语料库，更多的语料库可以从 OpenSLR（http://www.openslr.org）网站下载。OpenSLR 是一个致力于托管语音和语言资源（如用于语音识别的训练语料库和与语音识别相关的软件）的站点。

3.10.1　TIMIT 语音库

TIMIT 语音库有着准确的音素标注，因此可以应用于语音分割性能评价，同时该数据库又含有几百个说话人语音，所以也是评价说话人识别常用的权威语音库。

阅读语音的 TIMIT 语料库旨在提供声音语音数据和自动语音识别系统的开发和评估。TIMIT 包含 630 个说话人的宽带录音，8 种主要方言的美式英语，每种阅读 10 个语音丰富的句子。TIMIT 语料库包括时间对齐的单词内容，语音和单词转录以及每个话语的 16 位，16kHz 语音波形文件。语料库设计是麻省理工学院（MIT），斯坦福研究所（SRI）和德州仪器（TI）的共同努力。演讲在 TI 录制，转录于麻省理工学院，并由美国国家标准技术研究所（NIST）验证。

3.10.2　中文语音库

中文的语音识别公共数据集一共有 3 个，分别是：

- gale_mandarin：中文新闻广播数据集（LDC2013S08）。
- hkust：中文电话数据集（LDC2005S15, LDC2005T32）。
- thchs30：清华大学 30 小时数据集。

有些数据集中包含链接。如果需要复制链接文件下的内容，可以使用 cp -av 命令。

3.11　本章小结

自然语音处理的研究方向主要包括文本朗读、语音合成、语音识别等。语音识别技术主要包括特征提取技术、模式匹配准则及模型训练技术 3 个方面。本章详细介绍了语音识别的过程。

为了能开发出有效的语音识别系统，2009 年 Kaldi 在约翰霍普金斯大学诞生了。Kaldi 不是一个语音识别系统，而是一个建立语音识别系统的系统。Kaldi 使用运行于 Linux 操作系统的 C++、Perl、Python、Bash 等多种语言开发。Kaldi 这个名字来源于一名传说中的埃塞俄比亚的牧羊人，他发现了咖啡这种植物。

FFmpeg 是一个开源的软件项目，包含用于处理多媒体数据的库和程序。FFmpeg 项目是由法国人 Fabrice Bellard 在 2000 年发起的，此人也是著名的 CPU 模拟器项目 QEMU 的发起者，同时还是圆周率算法纪录的保持者。FFmpeg 中的 FF 是 Fast Forward 的意思，翻译成中文是“快进”。

可以采用动态时间归整算法识别孤立词语音。

第 4 章

Elasticsearch 分布式搜索引擎

Elasticsearch 是一个开源的全文搜索引擎，很多用户对于大规模集群应用时遇到的各种问题难以分析处理，本章主要介绍搭建 Elasticsearch 集群、索引数据、实现搜索接口、Elasticsearch 源代码分析等。

4.1 搭建 Elasticsearch 集群

在 Ubuntu 下，可以使用 debian 安装包安装 Elasticsearch。这个安装包仅依赖 Java。

使用 wget 命令下载 debian 安装包：

```
    # wget https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-6.5.4.deb
```

安装这个软件包：

```
    # dpkg -i elasticsearch-6.5.4.deb
```

安装成功后，可以使用 systemd 运行 Elasticsearch，启动服务：

```
    # sudo /bin/systemctl daemon-reload
    # sudo /bin/systemctl enable elasticsearch.service
    # sudo systemctl start elasticsearch.service
```

可以使用 curl 命令与 Elasticsearch 打交道。通过 curl 命令发送 HTTP 请求给本地节点的例子：

```
    # curl -XGET 'http://localhost:9200/'
```

返回类似下面的结果：

```
{
  "name" : "gjqSBPz",
  "cluster_name" : "elasticsearch",
  "cluster uuid" : "a6VrjsiaQbSZY0RWxfWNYw",
  "version" : {
    "number" : "6.5.4",
    "build flavor" : "default",
    "build_type" : "deb",
    "build hash" : "d2ef93d",
    "build date" : "2018-12-17T21:17:40.758843Z",
    "build snapshot" : false,
    "lucene version" : "7.5.0",
    "minimum_wire_compatibility_version" : "5.6.0",
    "minimum index compatibility version" : "5.0.0"
  },
  "tagline" : "You Know, for Search"
}
```

使用 curl 命令执行简单搜索请求：

```
# curl -XPOST "http://localhost:9200/ search" -H "Content-Type: application/
json" -d'
  {
     "query": {
       "match all": {}
     }
  }'
```

返回类似如下的结果：

```
{"took":35,"timed out":false," shards":{"total":5,"successful":5,"skipped
":0,"failed":0},"hits":{"total":0,"max_score":null,"hits":[]}}
```

默认情况下，Elasticsearch 的 RESTful 服务只有本机才能访问。也就是说，无法从主机访问虚拟机中的服务。为了方便调试，可以修改 config/elasticsearch.yml 文件，加入以下两行：

```
http.host: 0.0.0.0
transport.host: 127.0.0.1
echo >> ./elasticsearch.yml http.host: 0.0.0.0
echo >> ./elasticsearch.yml transport.host: 127.0.0.1
```

然后重启服务：

```
# sudo systemctl restart elasticsearch.service
```

但线上环境切忌不要这样配置，否则任何人都可以通过这个接口修改 ES 的数据。

可以使用_analyze 接口来分析某个 analyzer/tokenizer 如何分析和索引一段文字。使用 curl 命令测试 standard 分析器：

```
#  curl  -XGET  'http://localhost:9200/ analyze?pretty'  -H  'Content-Type:
application/json' -d'
{
  "analyzer" : "standard",
```

```
  "text" : "test 111 times"
}'
```

返回结果：

```
{
  "tokens" : [
    {
      "token" : "test",
      "start offset" : 0,
      "end offset" : 4,
      "type" : "<ALPHANUM>",
      "position" : 0
    },
    {
      "token" : "111",
      "start_offset" : 5,
      "end offset" : 8,
      "type" : "<NUM>",
      "position" : 1
    },
    {
      "token" : "times",
      "start_offset" : 9,
      "end offset" : 14,
      "type" : "<ALPHANUM>",
      "position" : 2
    }
  ]
}
```

4.2　索引数据

在 build.gradle 文件中增加 Elasticsearch 相关的依赖项：

```
dependencies {
    compile group: 'org.elasticsearch', name: 'elasticsearch', version: '6.5.4'
    compile
'org.elasticsearch.client:elasticsearch-rest-high-level-client:6.5.4'
    testCompile group: 'org.elasticsearch.test', name: 'framework', version:
'6.5.4'
    testCompile group: 'junit', name: 'junit', version: '4.11'
}
```

把配置信息保存到 config.properties 文件：

```
elasticsearch.host=localhost
elasticsearch.port=9200
```

可以使用 Apache Commons Configuration 读入配置信息。为 build.gradle 文件增加依赖项：

```
    dependencies {
        compile  group:  'org.apache.commons',  name:  'commons-configuration2',
version: '2.4'
        compile group: 'commons-beanutils', name: 'commons-beanutils', version:
'1.9.3'
    }
```

连接测试：

```
    Configurations configs = new Configurations();
    Configuration config = configs.properties(new File("config.properties"));

    String esHost = config.getString("elasticsearch.host");  //从配置文件读取字符串

    int port = config.getInt("elasticsearch.port",9200);   //从配置文件读取整数
    RestHighLevelClient client = new RestHighLevelClient(
          RestClient.builder(
                new HttpHost(esHost, port, "http")));

    MainResponse response = client.info(RequestOptions.DEFAULT);

    ClusterName clusterName = response.getClusterName();
    System.out.println( "集群名: " + clusterName);
    String clusterUuid = response.getClusterUuid();
    System.out.println( "集群 Uuid: " + clusterUuid);
    String nodeName = response.getNodeName();
    System.out.println( "节点名: " + nodeName);

    org.elasticsearch.Version version = response.getVersion();
    System.out.println( "版本号: " + version);

    client.close();
```

使用 spring-context 实现依赖注入。为 build.gradle 文件增加依赖项：

```
    dependencies {
        compile group: 'org.springframework', name: 'spring-context', version:
'5.1.5.RELEASE'
    }
```

实现依赖注入的 Crawler2Es 类如下：

```
    @Configuration
    public class Crawler2Es {
        @Bean(name="searchService")
        public RestHighLevelClient restHighLevelClient() throws Configuration
Exception {
            Configurations configs = new Configurations();
            org.apache.commons.configuration2.Configuration config =
                    configs.properties(new File("config.properties"));

            String esHost = config.getString("elasticsearch.host");
```

```
        int port = config.getInt("elasticsearch.port",9200);
        return  new RestHighLevelClient(
                RestClient.builder(new HttpHost(esHost, port, "http")));
    }

    public static void main(String[] args) throws IOException, ConfigurationException
    {
        ConfigurableApplicationContext context =
                new AnnotationConfigApplicationContext(
                        Crawler2Es.class);

        RestHighLevelClient client =
            context.getBean(RestHighLevelClient.class,"searchService");

        MainResponse response = client.info(RequestOptions.DEFAULT);

        ClusterName clusterName = response.getClusterName();
        System.out.println( "集群名: " + clusterName);
        String clusterUuid = response.getClusterUuid();
        System.out.println( "集群 Uuid: " + clusterUuid);
        String nodeName = response.getNodeName();
        System.out.println( "节点名: " + nodeName);

        org.elasticsearch.Version version = response.getVersion();
        System.out.println( "版本号: " + version);
        client.close();
        context.close();
    }
}
```

为了能够使用 Jsoup 解析网页，增加如下依赖项：

```
dependencies {
    compile group: 'org.jsoup', name: 'jsoup', version: '1.11.3'
}
```

使用 RestHighLevelClient 索引数据：

```
    String url = "http://mil.news.sina.com.cn/china/2018-11-06/doc-ihmutuea7527105.
shtml";
    Document doc = Jsoup.connect(url).get(); //解析的结果就是一个文档对象
    String title = doc.getElementsByClass("main-title").first()
          .text();
    String content = doc.getElementById("article").text();

    //构造一个 RestHighLevelClient 实例连接指定 IP 地址和端口号的 Elasticsearch 搜索服务
    RestHighLevelClient client = new RestHighLevelClient(
            RestClient.builder(new HttpHost(esHost, port, "http")));

    String index = "news";
    String type = "mil";
    String id = url;
```

```
    XContentBuilder b = XContentFactory.jsonBuilder().startObject();
    b.field("title", title);
    b.field("body", content);
    b.endObject();

    //把待索引内容从 XContent 转换到 JSON 格式
    String json = Strings.toString(b);

    IndexRequest request2 = new IndexRequest(index,  type,id );
    request2.source(json, XContentType.JSON);
    client.index(request2, RequestOptions.DEFAULT);

    client.close();
```

为了检查数据，可以用如下命令列出 Elasticsearch 服务器上的所有索引：

```
    #curl http://localhost:9200/_aliases?pretty=true
```

如果有大量数据需要使用集群建立索引，索引建立后，增量较少，则可以通过告知群集将指定的一个节点从分配中排除来停用节点。

```
    #curl -XPUT localhost:9200/_cluster/settings -H 'Content-Type: application/
json' -d '{
      "transient" :{
         "cluster.routing.allocation.exclude._ip" : "10.0.0.1"
       }
    }';echo
```

这将导致 Elasticsearch 将该节点上的分片分配给其余节点，而不会将群集状态更改为黄色或红色（即使分片只有 0 个副本）。

重新分配所有分片后，可以关闭节点并执行所需要执行的任何操作。完成后，包括要分配的节点，Elasticsearch 将再次重新平衡分片。

重新路由命令允许手动更改群集中各个分片的分配。例如，可以显式地将分片从一个节点移动到另一个节点，可以取消分配，并且可以将未分配的分片显式分配给特定节点。

以下是一个 reroute API 调用的简短示例：

```
    #curl  -X  POST  "localhost:9200/_cluster/reroute"  -H  'Content-Type:
application/json' -d'
    {
       "commands" : [
           {
              "move" : {
                 "index" : "test", "shard" : 0,
                 "from node" : "node1", "to node" : "node2"
              }
           },
           {
             "allocate replica" : {
                 "index" : "test", "shard" : 1,
```

```
                "node" : "node3"
            }
          }
      ]
  }
  '
```

4.3　实现搜索接口

最简单地，可以使用 QueryBuilders.matchAllQuery()方法查询所有数据：

```
QueryBuilder qb = QueryBuilders.matchAllQuery();
String index = "news"; //索引名

RestHighLevelClient client = new RestHighLevelClient(
      RestClient.builder(new HttpHost(Config.ip, 9200, "http")));

SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
sourceBuilder.query(qb);

SearchRequest request = new SearchRequest(index);
request.source(sourceBuilder);
SearchResponse searchResponse = client.search(request, RequestOptions.DEFAULT);

SearchHits hits = searchResponse.getHits();
for (SearchHit hit : hits) {
   Map<String, Object> retMap2 = hit.getSourceAsMap();
   //输出查询结果
   for (final Entry<String, Object> entry : retMap2.entrySet()) {
      System.out.println(entry.getKey() + " : " + entry.getValue());
   }
   System.out.println("score: " + hit.getScore()); //输出文档和查询词的匹配度分值
}
//关闭客户端
client.close();
```

按关键词查询：

```
String keyWords = "DNA";
String titleField = "title";
String bodyField = "body";

//查询内容列
MatchPhraseQueryBuilder pqBody =
       QueryBuilders.matchPhraseQuery(bodyField, keyWords);

//查询标题列
MatchPhraseQueryBuilder pqTitle =
```

```
            QueryBuilders.matchPhraseQuery(titleField, keyWords);

    //模糊查询
    QueryStringQueryBuilder fuzzyQb =
            new QueryStringQueryBuilder(keyWords);

    //或者条件连接上面3个查询
    QueryBuilder qb =
            QueryBuilders.boolQuery().should(pqBody).should(pqTitle).should
(fuzzyQb);
```

假设按20条记录分页，显示第2页的实现如下：

```
    SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
    sourceBuilder.query(qb);

    int from = 10;
    int size = 20;

    //设置确定结果索引的from选项以开始搜索
    //默认为0
    sourceBuilder.from(from);

    //设置size选项，确定要返回的搜索命中数
    //默认为10
    sourceBuilder.size(size);

    SearchRequest request = new SearchRequest(index);
    request.source(sourceBuilder);
    SearchResponse searchResponse = client.search(request, RequestOptions.DEFAULT);

    SearchHits hits = searchResponse.getHits();

    long totalHits = hits.getTotalHits(); //得到结果总数

    System.out.println("totalsize" + String.valueOf(totalHits));

    if (hits.totalHits > (from + size)) {  //判断是否有更多结果可以显示
       System.out.println("还有更多匹配结果");
    } else {
       System.out.println("结果显示完毕");
    }
```

4.4 搜索界面开发

OpenAPI规范（OAS）定义了REST API的标准，与编程语言无关的接口描述，允

许人与计算机发现和理解服务的功能，而无需访问源代码，附加文档或检查网络流量。通过 OpenAPI 正确定义后，使用者可以使用最少量的实现逻辑来理解远程服务并与之交互。与低级编程的接口描述类似，OpenAPI 规范消除了调用服务时的猜测。

OpenAPI 生成器（https://github.com/OpenAPITools/openapi-generator）允许在给定 OpenAPI 规范（支持 2.0 和 3.0）的情况下自动生成 API 客户端库（SDK 生成），服务器存根、文档和配置。

OpenAPI 生成器支持的 API 客户端包括 C#、Java、Perl、PHP、Python、Typescript，支持的服务器存根包括 C#(ASP.NET Core、NancyFx)，Java(MSF4J、Spring、Undertow、JAX-RS: CDI、CXF、Inflector、RestEasy、Play Framework、PKMST),Kotlin(Spring Boot),PHP(Laravel、Lumen、Slim、Silex、Symfony、Zend Expressive),Python(Flask),NodeJS,Ruby(Sinatra、Rails5),Rust(rust- server),Scala(Finch、Lagom、Scalatra)。

这里分别介绍使用 Spring Boot 和 ASP.Net 开发搜索界面。

4.4.1 使用 Spring Boot 开发搜索界面

可以使用 Spring Boot 轻松创建基于微服务的搜索引擎界面。微服务可以建立在基于请求和响应的 REST（表述性状态转移）协议之上。REST 是一种针对网络应用的设计和开发方式，可以降低开发的复杂性，提高系统的可伸缩性的协议。

可以从引导器网站 https://start.spring.io/生成出适当的文件夹结构。HATEOAS(the hypermedia as the engine of application statue)是 REST 架构的主要约束。为了支持超文本驱动的 REST Web 服务表示，生成项目时，可以选择 HATEOAS 依赖项。

或者使用 curl 命令生成使用 Gradle 构建的项目：

```
    #curl "https://start.spring.io/starter.zip?type=gradle-project&language=
java&bootVersion=2.1.2.RELEASE&baseDir=demo&groupId=com.lietu&artifactId=demo
&name=demo&description=Demo+project+for+Spring+Boot&packageName=com.example.d
emo&packaging=jar&javaVersion=1.8&autocomplete=&generate-project=&style=hateo
as" -H "User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64; rv:65.0) Gecko/20100101
Firefox/65.0" -H "Accept: text/html,application/xhtml+xml,application/ xml;q=0.9,
image/webp,*/*;q=0.8" -H "Accept-Language: en-US,en;q=0.5" --compressed -H
"Referer: https://start.spring.io/" -H "Connection: keep-alive" -H "Cookie:
  cfduid=d51277f2ec7b7eacd5eb4737f9125c0a61549411878;
 ga=GA1.2.1439080563.1549411881; gid=GA1.2.1544543470.1549411881; gat UA-2728886-
24=1" -H "Upgrade-Insecure-Requests: 1" -H "TE: Trailers" --output demo.zip
```

可以把生成出来的项目导入 Eclipse。

为了加快下载速度，可以在 build.gradle 文件中增加仓库地址：

```
    repositories {
       maven {url 'http://maven.aliyun.com/nexus/content/groups/public/'}
       mavenCentral()
```

```
    }
```

创建一个 Example 类：

```
    package com.lietu.demo;

    import org.springframework.boot.*;
    import org.springframework.boot.autoconfigure.*;
    import org.springframework.boot.builder.SpringApplicationBuilder;
    import
org.springframework.boot.web.servlet.support.SpringBootServletInitializer;
    import org.springframework.web.bind.annotation.*;

    @RestController
    @SpringBootApplication
    public class Example extends SpringBootServletInitializer{
        @Override
        protected SpringApplicationBuilder configure(SpringApplicationBuilder
application) {
              return application.sources(Example.class);
        }

       @RequestMapping("/")
       String home() {
          return "Hello World!";
       }

       public static void main(String[] args) {
          SpringApplication.run(Example.class, args);
       }

    }
```

运行 Example 类。在另外一个控制台查看启动的服务：

```
    # http http://localhost:8080/
```

这时，Sprinboot 使用嵌入的 Tomcat 提供 Web 服务。

src\main\resources\application.properties 文件包含所有配置相关的信息。例如修改端口号为 8090：

```
    server.port = 8090
```

可以使用 bootJar 任务构建可执行的 jar：

```
    bootJar {
        mainClassName = 'com.lietu.demo.Example'
    }
```

要构建可执行的 jar，可以执行以下命令：

```
    >.\gradlew bootJar
```

生成的可执行归档文件将放在 build\libs 目录中。在部署时，直接使用 java -jar demo-0.0.1-SNAPSHOT.jar 进行启动。

如果使用外部的 Web 服务器，可以使用 bootWar 构建 War 包。首先需要把：

```
apply plugin: 'java'
```

修改成：

```
apply plugin: 'war'
```

使用 bootWar 任务构建 War 包：

```
bootWar {
    mainClassName = 'com.lietu.demo.Example'
}
```

将 build.gradle 文件中的依赖项修改成如下的形式：

```
dependencies {
    implementation('org.springframework.boot:spring-boot-starter-web')
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'
   testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

最后下载 Tomcat：

```
#wget http://mirrors.shu.edu.cn/apache/tomcat/tomcat-9/v9.0.14/bin/apache-
tomcat-9.0.14.tar.gz
```

把生成出来的 demo-0.0.1-SNAPSHOT.war 部署于这个 Tomcat 解压缩后的 webapps 目录。

也可以使用 Jetty(https://github.com/eclipse/jetty.project)作为 Web 服务器。

将 build.gradle 文件中的依赖项修改成如下的形式：

```
dependencies {
    implementation('org.springframework.boot:spring-boot-starter-web')
    providedRuntime 'org.springframework.boot:spring-boot-starter-jetty'
   testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

使用默认选项运行 Jetty：

```
> cd demo-base
>java -jar ../start.jar
```

静态页面或者 JS、CSS、图片之类的静态资源可以放入 resources 目录下的 static 目录。把搜索首页 index.html 放入 static 目录。

```
<html>
<head>
  <title>猎兔搜索</title>
  <META http-equiv=Content-Type content="text/html; charset=utf-8">
</head>
<body>
<form id="sform" method="get" action="search">
                <input  type="text" name="query"  />
                < input  type=submit value=猎兔搜索>
</form>
```

```
 </body>
</html>
```

把 Example 类修改如下：

```
@RestController
@SpringBootApplication
public class Example extends SpringBootServletInitializer {
   @Override
   protected  SpringApplicationBuilder  configure(SpringApplicationBuilder
application) {
      return application.sources(Example.class);
   }

   @RequestMapping("/search")
   public String search(@RequestParam(value = "query") String q) {
      return "Get query:"+q;
   }

   public static void main(String[] args) {
      SpringApplication.run(Example.class, args);
   }

}
```

这里使用了 RequestParam 注解把 URL 中的参数值解析到方法中的参数。

模型-视图-控制器（MVC）软件设计模式是一个用于在软件应用程序内分离关注点的方法。原则上，应用程序逻辑或控制器与用于向用户或视图层显示信息的技术分离。模型是控制器和视图层之间的通信工具。

在应用程序内，视图层可以使用一种或多种不同的技术来呈现视图。SpringBoot 基于 Web 的应用程序支持各种视图选项，通常称为视图模板。这些技术被描述为“模板”，因为它们提供了一种标记语言，用于在服务器端呈现期间公开视图中的模型属性。SpringBoot 支持多种模板引擎。这里介绍使用 Freemarker(https://github.com/apache/freemarker/)模板引擎。

模板引擎 FreeMarker 用于根据模板生成文本输出，可以用于生成 HTML 网页或者自动生成源代码。它是一个 Java 包，是 Java 程序员的类库。它本身并不是最终用户的应用程序，而是程序员可以嵌入到他们的产品中的东西。Freemarker 旨在用于生成 HTML Web 页面，特别是遵循 MVC 模式的基于 Servlet 的应用程序。

管理项目的 build.gradle 内容如下：

```
apply plugin: 'java'
apply plugin: 'eclipse'
archivesBaseName = 'Concretepage'
version = '1.0-SNAPSHOT'
repositories {
    maven { url "https://repo.spring.io/libs-release" }
    mavenLocal()
```

```
        mavenCentral()
    }

    dependencies {
        compile 'org.freemarker:freemarker:2.3.28'
    }
```

Freemarker 模板文件 search-template.ftl 的内容如下。

```
<html>
<head><title> ${websiteTitle} </title>
<body>
<p>
  ${message}
</p>
</body>
</html>
```

此模板将生成带有标题和正文消息的 HTML 输出。使用 search-template.ftl 生成 HTML 源代码：

```
//取得模板相对路径
File r=new File("");
String templatePath=r.getAbsolutePath() + "/templates";

//实例化配置类
Configuration cfg = new Configuration(Configuration.VERSION_2_3_28);
cfg.setDirectoryForTemplateLoading(new File(templatePath));
cfg.setDefaultEncoding("UTF-8");
cfg.setTemplateExceptionHandler(TemplateExceptionHandler.RETHROW_HANDLER);

//创建数据模型
Map<String, Object> modelMap = new HashMap<>();
modelMap.put("websiteTitle", "Freemarker Template Demo");
modelMap.put("message",
     "Getting started with Freemarker.<br/>Find a simple Freemarker demo.");
//实例化模板
Template template = cfg.getTemplate("search-template.ftl");

//将 freemarker 输出写入 StringWriter
StringWriter stringWriter = new StringWriter();
template.process(modelMap, stringWriter);
//从 StringWriter 获取字符串
String content = stringWriter.toString();
System.out.println(content);

//控制台输出
Writer console = new OutputStreamWriter(System.out);
template.process(modelMap, console);
console.flush();

//文件输出
```

```
    Writer file = new FileWriter (new File(templatePath+"/search-template-output.
html"));
    template.process(modelMap, file);
    file.flush();
    file.close();
```

接下来通过遍历列表显示返回的搜索结果。例如，后台传入一个 uerList 的 list 或者 Set，在模板中显示如下：

```
    <#list userList as user>
      ${user.name}
    </#list>
```

首先定义实体类 WebItem：

```
    public class WebItem {
       private String url;
       private String title;
       private String body;

       public WebItem(String url, String title) {
          this.url = url;
          this.title = title;
       }
       public WebItem(String url, String title,String content) {
          this.url = url;
          this.title = title;
          body = content;
       }

       public String getUrl() {
          return url;
       }
       public void setUrl(String url) {
          this.url = url;
       }
       public String getTitle() {
          return title;
       }
       public void setTitle(String title) {
          this.title = title;
       }

       public String getBody() {
          return body;
       }
       public void setBody(String body) {
          this.body = body;
       }
    }
```

将 search-template.ftl 文件修改成：

```
    <html>
```

```
<head><title> ${websiteTitle} </title>
<body>
<input value="${query}"/>
Results
<#list resultItems as result>
    ${result index + 1}. <a href="${result.url}"> ${result.title} </a> <br/>
</#list>
</body>
</html>
```

将使用 search-template.ftl 生成的 HTML 代码修改成：

```
Map<String, Object> map = new HashMap<>();
map.put("websiteTitle", "Freemarker Template Demo");
map.put("query", "keyword");
List< WebItem> items = new ArrayList<>();
items.add(new WebItem("http://url1.com","URL One"));
items.add(new WebItem("http://url2.com","URL Two"));
items.add(new WebItem("http://url3.com","URL Three"));
map.put("resultItems", items);
```

生成的 HTML 源代码如下：

```
<html>
<head><title> Freemarker Template Demo </title>
<body>
<input value="keyword"/>
Results
    1. <a href="http://url1.com"> URL One </a> <br/>
    2. <a href="http://url2.com"> URL Two </a> <br/>
    3. <a href="http://url3.com"> URL Three </a> <br/>
</body>
</html>
```

当无法从文件加载模板时，可以手动创建模板。例如：

```
String templateStr = "Hello ${user}";
//根据 StringReader 构建模板
Template template = new Template("path", new StringReader(templateStr),
      new Configuration(Configuration.VERSION 2 3 28));

Map<String, Object> map = new HashMap<>();
map.put("user", "test");

StringWriter stringWriter = new StringWriter();
template.process(map, stringWriter);
String content = stringWriter.toString();
System.out.println(content);
```

可以使用 Freemarker 中的宏实现翻页。

先通过例子了解模板中宏的用法。

首先在 macros.ftl 文件中定义宏 macro1：

```
<#macro macro1>
```

```
Helloworld.
</#macro>
```

然后在 testMacro.ftl 文件中使用宏：

```
<#import "./macros.ftl" as my>
<@my.macro1 />
```

使用 testMacro.ftl 模板：

```
Configuration cfg = new Configuration(Configuration.VERSION_2_3_28);
cfg.setDirectoryForTemplateLoading(new File(templatePath));
cfg.setDefaultEncoding("UTF-8");
cfg.setTemplateExceptionHandler(TemplateExceptionHandler.RETHROW_HANDLER);

Template template = cfg.getTemplate("testMacro.ftl");

//控制台输出
Writer console = new OutputStreamWriter(System.out);
template.process(null, console);
console.flush();
```

通过 JavaScrip 实现的分页宏：

```
<#-- 自定义的分页指令
属性：
pageNo       当前页号(int 类型)
pageSize     每页要显示的记录数(int 类型)
pageCount    总页数
toURL        单击分页标签时要跳转到的目标 URL(string 类型)
-->
<#macro pager pageNo pageSize toURL pageCount>
   <#-- 输出分页表单 -->
   <div class="page">
      <form method="post" action="${toURL}" name="qPagerForm">
        <#--用于记录当前的页数 -->
        <input type='hidden' name ='page' />
        <#-- 上一页处理 -->
        <#if (pageNo == 1)>
        <span class="disabled"><a >上一页</a></span>
        <#else>
        <span ><a href="javascript:page(${pageNo - 1})">上一页</a></span>
        </#if>
        <#-- 如果前面页数过多,显示... -->
        <#assign start=1>
        <#if (pageNo > 4)>
            <#assign start=(pageNo - 2)>
            <a href="javascript:page(1)">1</a>
            <span style='color:#444693;'>...</span>
        </#if>
        <#-- 显示当前页号和它附近的页号 -->
        <#assign end=(pageNo + 2)>
```

```
            <#if (end > pageCount)>
                <#assign end=pageCount>
            </#if>
            <#list start..end as i>
                <#if (pageNo==i)>
                    <span ><a  class="dq">${i}</a></span>
                <#else>
                   <span><a href="javascript:page(${i})">${i}</a></span>
                </#if>
            </#list>
            <#-- 如果后面页数过多,显示... -->
            <#if (end < pageCount - 1)>
                <span style='color:#444693;'>...</span>
            </#if>
            <#if (end < pageCount)>
                <a href="javascript:page(${pageCount})">${pageCount}</a>
            </#if>
            <#-- 下一页处理 -->
            <#if (pageNo == pageCount)>
                <span class="disabled"><a >下一页</a></span>
            <#else>
                <span>
                <a href="javascript:page(${pageNo + 1})">下一页</a>
                </span>
            </#if>
        </form>
        <script language="javascript">
          function page(no){
                var $qForm=$("form[name='qPagerForm']");
                $qForm.find("input[name='page']").val(no);
                $qForm.submit();
          }
        </script>
    </div>
</#macro>
```

使用这个宏的模板 testpager.ftl 的内容如下：

```
<#import "pager.ftl" as page />
<@page.pager pageNo=p.pageNo pageSize=p.pageSize toURL=p.toURL pageCount=p.
pageCount />
```

测试宏：

```
Configuration cfg = new Configuration(Configuration.VERSION 2 3 28);
cfg.setDirectoryForTemplateLoading(new File(templatePath));
cfg.setDefaultEncoding("UTF-8");
cfg.setTemplateExceptionHandler(TemplateExceptionHandler.RETHROW HANDLER);

//创建数据模型
Map<String, Object> modelMap = new HashMap<>();

Map<String, Object> paginationData = new HashMap<>();
```

```
    paginationData.put("pageNo", 5);
    paginationData.put("pageSize", 10);
    paginationData.put("pageCount", 30);
    paginationData.put("toURL", "http://www.lietu.com");

    modelMap.put("p", paginationData);

    //实例化模板
    Template template = cfg.getTemplate("testpager.ftl");

    //将 freemarker 输出写入 StringWriter
    StringWriter stringWriter = new StringWriter();
    template.process(modelMap, stringWriter);
    //从 StringWriter 获取 String
    String content = stringWriter.toString();
    System.out.println(content);
```

输出结果如下：

```
<div class="page">
   <form method="post" action="http://www.lietu.com" name="qPagerForm">
      <input type='hidden' name ='page' />
      <span ><a href="javascript:page(4)">上一页</a></span>
         <a href="javascript:page(1)">1</a>
         <span style='color:#444693;'>...</span>
             <span><a href="javascript:page(3)">3</a></span>
             <span><a href="javascript:page(4)">4</a></span>
             <span ><a  class="dq">5</a></span>
             <span><a href="javascript:page(6)">6</a></span>
             <span><a href="javascript:page(7)">7</a></span>
         <span style='color:#444693;'>...</span>
         <a href="javascript:page(30)">30</a>
         <span>
         <a href="javascript:page(6)">下一页</a>
         </span>
   </form>
   <script language="javascript">
      function page(no){
         var $qForm=$("form[name='qPagerForm']");
         $qForm.find("input[name='page']").val(no);
         $qForm.submit();
      }
   </script>
</div>
```

或者使用自定义的 PagerTag 类实现翻页。PagerTag 代码如下：

```
public class PagerTag {
   static String
      charset = "UTF-8";  //字符集

   static final String
```

```
        DEFAULT_ID = "pager";            //标签 ID

    private static final int
        DEFAULT_MAX_ITEMS = Integer.MAX_VALUE,
        DEFAULT_MAX_PAGE_ITEMS = 10,
        DEFAULT_MAX_INDEX_PAGES = 10;

    static final String
        OFFSET_PARAM = ".offset";        //偏移量参数

    /*
     * 标签属性
     */
    private String url = null;
    private String index = null;
    private int items = 0;
    private int maxItems = DEFAULT_MAX_ITEMS;
    private int maxPageItems = DEFAULT_MAX_PAGE_ITEMS;  //每页显示结果数
    private int maxIndexPages = DEFAULT_MAX_INDEX_PAGES;
    private boolean isOffset = false;
    private String export = null;

    /*
     * 标签变量
     */
    private StringBuffer uri = null;
    private int params = 0;
    private int offset = 0;

    public void setOffset(int offset) {
        this.offset = offset;
    }

    private long itemCount = 0;  //条目总数

    private int pageNumber = 0;
    private Integer pageNumberInteger = null;

    private String idOffsetParam = DEFAULT_ID+OFFSET_PARAM;

    public PagerTag(String u) {
        this.url = u;
        startTag();
    }

    public PagerTag(String u,long totalHits) {
        this.url = u;
        itemCount = totalHits;
        System.out.println("this.url :" + this.url);
        startTag();
```

```
    }

    public void startTag(){
        String baseUri=null;
        if (url != null) {
            baseUri = url;
        }
        //TODO 增加从 HttpServletRequest 对象取得参数

        if (uri == null)
            uri = new StringBuffer(baseUri.length() + 32);
        else
            uri.setLength(0);
        uri.append(baseUri);

    }

}
```

测试 PagerTag：

```
int totalHits = 132;            //结果总数
String keyWords = "DNA";        //查询词

PagerTag pagerTag = new PagerTag("./search", totalHits);

int from = 40;                  //返回结果的起始偏移量
int size = 20;                  //最多返回结果数量

pagerTag.setOffset(from);
pagerTag.setMaxPageItems(size);
pagerTag.addParam("query", keyWords);

System.out.println(pagerTag.hasNextPage());      //是否还有下一页结果
System.out.println(pagerTag.getMaxPageItems());  //得到最多返回结果数量
System.out.println(pagerTag.getNextUrl());       //下一页网址
```

在数据模型中使用 PagerTag：

```
PagerTag pagerTag =  new PagerTag("./search",totalHits);

pagerTag.setOffset(from);
pagerTag.setMaxPageItems(size);
pagerTag.addParam("query", keyWords);

modelMap.put("pager", pagerTag);
```

在模板中使用 pager 对象。

```
<#if (pager.hasNextPage())>
    <a href="${pager.getNextUrl()}" class="rnavLink">下一页 &#187;</a>
</#if>
```

为了输出翻页链接，这里调用了 pager 对象的 hasNextPage()方法和 getNextUrl()方法。实际输出的网页样例如下：

```
<a href="./search?query=%E5%A3%AB%E5%85%B5&amp;pager.offset=60" class="rnavLink">
下一页 &#187;</a>
```

为了测试 SpringBoot 集成 Freemarker 模板引擎，首先修改 build.gradle，增加对 spring-boot-starter-freemarker 的引用：

```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-freemarker'
    implementation 'org.springframework.boot:spring-boot-starter-hateoas'
    implementation 'org.springframework.boot:spring-boot-starter-web'
    providedRuntime 'org.springframework.boot:spring-boot-starter-tomcat'

    testImplementation 'org.springframework.boot:spring-boot-starter-test'
}
```

将 SpringBoot 的控制器类 Example 的 home()方法修改成如下的形式：

```
@RequestMapping("/")
String home() throws TemplateException, IOException {
    String templateStr = "Hello ${user}";
    Template template = null;
    template = new Template("name", new StringReader(templateStr),
            new Configuration(Configuration.VERSION_2_3_28));

    Map<String, Object> map = new HashMap<>();
    map.put("user", "tom");

    StringWriter stringWriter = new StringWriter();
    template.process(map, stringWriter);
    String content = stringWriter.toString();
    System.out.println(content);

    return content;
}
```

执行 Example 类检查网站的输出。一切正常后可以把模板文件 search-template.ftl 复制到 src\main\resources\templates\目录。然后继续修改 home()方法：

```
@RequestMapping("/")
public ModelAndView home() throws TemplateException, IOException {
    ModelAndView mv = new ModelAndView("search-template");
    mv.getModelMap().addAttribute("websiteTitle", "Freemarker Template Demo");
   mv.getModelMap().addAttribute("message", "Getting started with Freemarker.
<br/>Find a simple Freemarker demo.");
    return mv;
}
```

执行 Example 类并检查网站的输出：

```
>curl http://localhost:8080/
<html>
```

```
<head><title> Freemarker Template Demo </title>
<body>
<p>
  Getting started with Freemarker.<br/>Find a simple Freemarker demo.
</p>
</body>
</html>
```

为了整合搜索首页，将 Example 类修改成如下：

```
@RestController
@SpringBootApplication
public class Example extends SpringBootServletInitializer{
    @Override
    protected  SpringApplicationBuilder  configure(SpringApplicationBuilder
application) {
         return application.sources(Example.class);
    }
   @RequestMapping("/search")
   public ModelAndView search(@RequestParam(value = "query") String q)
            throws TemplateException, IOException {
      ModelAndView mv = new ModelAndView("search-template");
      mv.getModelMap().addAttribute("websiteTitle",  "Freemarker  Template
Demo");
      mv.getModelMap().addAttribute("message",
             "Getting started with Freemarker.<br/> query:"+q);
      return mv;
   }
   public static void main(String[] args) {
      SpringApplication.run(Example.class, args);
   }

}
```

或者采用更直接的方式得到查询和翻页参数并在响应中设置 Cookie：

```
@RequestMapping(value = "/search", method = RequestMethod.GET)
public ModelAndView search(HttpServletRequest request,
             HttpServletResponse response)  throws IOException {
    response.setCookie(cookie);  //
    ModelAndView mav = new ModelAndView();
    String q = request.getParameter("query");
    int offset = Integer.parseInt(request.getParameter("pager.offset"));
    return mav;
}
```

假设 src\main\resources\templates\目录下的 bottom.html 内容如下：

```
<br><FONT style="FONT-SIZE: 14px"><CENTER>&copy;2019 Lietu</CENTER></FONT>
```

可以使用 include 指令插入这个文件：

```
<html>
<head><title> ${websiteTitle} </title>
<body>
<p>
```

```
      ${message}
    </p>
    <#include "/bottom.html">
    </body>
    </html>
```

在 CSS 目录下的 default.css 文件中定义了 CSS 样式。HTML 文档应用 CSS 样式后的 search.html 文件内容如下：

```
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://
www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

    <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
       <title>猎兔搜索引擎测试版本 v1</title>
       <link href="CSS/default.css" rel="stylesheet" type="text/css" />
    </head>
    <body>
    <form id="form1">
       <div id="sHeader">
          <div id="searcher">
            <form method="get" action="search">
                <input ID="tbKeyword" class="input text" />
                <input ID="btSearch" type=submit value="试着搜搜"/>
            </form>
          </div>
          </div>
       <div id="content">

          <div id="searcherContent">
             <div id="main">

                <div id="search-report">约有记录：XXXXXX 条</div>

                        <div class="search-item">
                           <b><a href='http://test' target="_blank">title</a>
</b>
                        </div>

                        <div class="search-item">
                           body
                        </div>

             </div>
          </div>

       </div>
      </form>
    </body>
    </html>
```

根据 search.html 得到的模板文件如下：

```
    <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.
w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
    <html xmlns="http://www.w3.org/1999/xhtml">
    <head>
       <title>猎兔搜索引擎测试版本 v1</title>
       <link href="CSS/default.css" rel="stylesheet" type="text/css" />
    </head>
    <body>
    <form id="sform">
       <div id="sHeader">
          <div id="searcher">
            <form method="get" action="search">
                <input ID="tbKeyword"  class="input text" value="${query}" />
                <input ID="btSearch" type=submit value="试着搜搜"/>
            </form>
          </div>
          </div>
       <div id="content">

          <div id="searcherContent">
             <div id="main">

                <div id="search-report">约有记录：${totalsize}条</div>

    <#list resultItems as result>
                        <div class="search-item">
                           <b><a href='${result.url}' target=" blank">${result.
title}</a></b>
                        </div>

                        <div class="search-item">
                           ${result.body}
                        </div>
    </#list>

             </div>
          </div>

       </div>
      </form>
    </body>
    </html>
```

Freemarker 集成 Elasticsearch 搜索的 search()方法如下：

```
    public static void search(Map<String, Object> modelMap) throws IOException {
       List<WebItem> items = new ArrayList<>();

       String titleField = "title";
       String bodyField = "body";

       MatchPhraseQueryBuilder pqBody =
```

```
                    QueryBuilders.matchPhraseQuery(bodyField, keyWords);

        MatchPhraseQueryBuilder pqTitle =
                    QueryBuilders.matchPhraseQuery(titleField, keyWords);

        QueryStringQueryBuilder fuzzyQb = new QueryStringQueryBuilder(keyWords);

        QueryBuilder qb =
              QueryBuilders.boolQuery().should(pqBody).should(pqTitle).should
(fuzzyQb);

        String index = "news"; //索引名

        RestHighLevelClient client =
             new RestHighLevelClient(RestClient.builder(new HttpHost(Config.ip,
9200, "http")));
        SearchRequest request = new SearchRequest(index);

        SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
        sourceBuilder.query(qb);
        request.source(sourceBuilder);

        HighlightBuilder highlightBuilder = new HighlightBuilder();
        HighlightBuilder.Field highlightTitle = new HighlightBuilder.Field
(titleField); //title 字段高亮

        highlightTitle.highlighterType("unified"); //配置高亮类型
        //如果索引中保留了位置偏移量 高亮类型 也可以是 fvh
        //highlightTitle.highlighterType("fvh");

        highlightBuilder.field(highlightTitle); //添加到 builder
        HighlightBuilder.Field highlightBody = new HighlightBuilder.Field
(bodyField);
        highlightBuilder.field(highlightBody);

        highlightBuilder.preTags("<span style=\"color:red\">");
        highlightBuilder.postTags("</span>");

        sourceBuilder.highlighter(highlightBuilder);

        SearchResponse searchResponse = client.search(request, RequestOptions.
DEFAULT);

        SearchHits hits = searchResponse.getHits();

        long totalHits = hits.getTotalHits(); //得到结果总数

        modelMap.put("query", keyWords);
        modelMap.put("totalsize", String.valueOf(totalHits));

        for (SearchHit hit : hits) {
```

```
            //键是列名，值是文档中该列的值
            Map<String, Object> result = hit.getSourceAsMap();
            WebItem webItem = new WebItem((String) result.get("url"),
                    (String) result.get("title"),
                    (String) result.get("body"));

            HighlightField  titleHighlight  =  hit.getHighlightFields().get
(titleField);

            if (titleHighlight != null) {
               Text[] text = titleHighlight.fragments();
               String fragmentString = StringUtils.join(text, "...");
               webItem.setTitle(fragmentString);
            }

            HighlightField  bodyHighlight  =  hit.getHighlightFields().get
(bodyField);

            if (bodyHighlight != null) {
               Text[] text = bodyHighlight.fragments();
               String fragmentString = StringUtils.join(text, "...");
               webItem.setBody(fragmentString);
            }

            items.add(webItem);
        }
        client.close();
        modelMap.put("resultItems", items);
    }
```

使用 search()方法创建数据模型：

```
    Map<String, Object> map = new HashMap<>();
    search(map);
```

FMParser 类由 JavaCC(https://github.com/javacc/javacc)从语法文件 FTL.jj 生成。JavaCC 从 EBNF 表示法编写的正规文法中生成解析器。

如果对类路径有 org.elasticsearch.client:elasticsearch-rest-high-level-client 依赖关系，Spring Boot 将自动配置一个 RestHighLevelClient，它包装任何现有的 RestClient bean，重用其 HTTP 配置。

要将 REST 客户端与 Spring Boot 集成，可以像任何其他依赖项一样创建配置 Bean，然后在任何需要的地方自动装配依赖项：

```
    @Configuration
    public class ElasticsearchConfig {

       @Value("${elasticsearch.host}")
       private String host;

       @Value("${elasticsearch.port}")
```

```
        private int port;

        @Bean
        public RestHighLevelClient restHighLevelClient() {
           return  new RestHighLevelClient(RestClient.builder(new HttpHost(host,
port, "http")));
        }
    }
```

application.properties 文件的内容如下：

```
    elasticsearch.host=localhost
    elasticsearch.port=9200
```

可以使用@ResponseBody 注解返回 JSON 格式的数据，而 ResponseEntity 则表示整个 HTTP 响应：状态代码、标头和正文。因此，可以使用它来完全配置 HTTP 响应。使用 ResponseEntity 返回查询结果的代码如下：

```
    @RestController
    @SpringBootApplication
    public class Example extends SpringBootServletInitializer {
       @Override
       protected  SpringApplicationBuilder  configure(SpringApplicationBuilder
application) {
          return application.sources(Example.class);
       }

        @Autowired
        private RestHighLevelClient client;  //按照类型装配依赖对象

       @RequestMapping("/search")
       public ResponseEntity<List<WebItem>>  search(@RequestParam(value = "query")
String q) throws IOException {
          //反序列化成 Java 动态数组对象
          List<WebItem> items = new ArrayList<WebItem>();

          String titleField = "title";
          String bodyField = "body";

          MatchPhraseQueryBuilder  pqBody  =  QueryBuilders.matchPhraseQuery
(bodyField, q);

          MatchPhraseQueryBuilder pqTitle = QueryBuilders.matchPhraseQuery
(titleField, q);

          QueryStringQueryBuilder fuzzyQb = new QueryStringQueryBuilder(q);

          QueryBuilder qb =
              QueryBuilders.boolQuery().should(pqBody).should(pqTitle).should
(fuzzyQb);

          String index = "news"; //索引名
```

```
        SearchRequest request = new SearchRequest(index);

        SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
        sourceBuilder.query(qb);
        request.source(sourceBuilder);

        HighlightBuilder highlightBuilder = new HighlightBuilder();
        HighlightBuilder.Field highlightTitle = new HighlightBuilder.Field
(titleField); //title 字段高亮
        //如果索引中保留了位置偏移量 高亮类型 也可以是 fvh
        highlightTitle.highlighterType("unified");      //配置高亮类型

        highlightBuilder.field(highlightTitle);         //添加到 builder
        HighlightBuilder.Field highlightBody = new HighlightBuilder.Field
(bodyField);
        highlightBuilder.field(highlightBody);
        sourceBuilder.highlighter(highlightBuilder);

        SearchResponse searchResponse = client.search(request, RequestOptions.
DEFAULT);

        SearchHits hits = searchResponse.getHits();

        long totalHits = hits.getTotalHits(); //得到结果总数

        for (SearchHit hit : hits) {
          Map<String, Object> result = hit.getSourceAsMap();
          //得到 URL 列
          String url = (String)result.get("url");
          //使用 WebItem bean 来构建响应
          WebItem item = new WebItem(url);

          HighlightField titleHighlight = hit.getHighlightFields().get
(titleField);

          if (titleHighlight != null) {
              Text[] text = titleHighlight.fragments();

              String fragmentString = StringUtils.join(text, "...");
              item.setTitle(fragmentString);
          }

          HighlightField bodyHighlight = hit.getHighlightFields().get
(bodyField);

          if (bodyHighlight != null) {
              Text[] text = bodyHighlight.fragments();

              String fragmentString = StringUtils.join(text, "...");
```

```
                item.setBody(fragmentString);
            }
            items.add(item);
        }

        return ResponseEntity.ok(items);
    }

    public static void main(String[] args) {
        SpringApplication.run(Example.class, args);
    }

}
```

这里将 search()方法直接放置于主类。也可以将 search()方法分离出来，单独在 SearchController 类中实现：

```
@RestController
public class SearchController {

    @Autowired
    private RestHighLevelClient client;

    @RequestMapping("/search")
    public ResponseEntity<List<WebItem>>  search(
            @RequestParam(value = "query") String q) throws IOException {
        //search 方法的具体实现
    }

}
```

缓存有助于通过减少数据库或任何昂贵资源之间的往返次数来提高应用程序的性能。但此时我们将面对像我们必须执行大量数据库查询的场景，并且假设数据库中的数据很少会发生变化。对于这种情况，每次都调用数据库不是一个好主意，但可以只缓存第一次调用数据库时的结果，并为其他调用再次返回相同的数据。

为了在 Spring Boot 应用程序中使用缓存，可以首先将@EnableCaching 注解写入主类，然后将@Cacheable 注解添加到要缓存结果的方法中。

例如，要缓存文章。表示文章的实体类 Article 如下：

```
public class Article implements Serializable {
    private long articleId;
    private String title;
    private String category;

    public Article(long i, String t, String c) {
        articleId = i;
        title = t;
        category = c;
    }
}
```

@EnableCaching 支持注解驱动的缓存管理功能。它负责注册所需的 Spring 组件以启用注解驱动的缓存管理。@EnableCaching 使用@Configuration 或@SpringBootApplication 进行注解。CacheExample 主类实现如下：

```
@RestController
@SpringBootApplication
@EnableCaching
public class CacheExample extends SpringBootServletInitializer {
   @RequestMapping("/get-article-info")
   @Cacheable(value="cacheArticleInfo")
   public ResponseEntity<List<Article>> articleInformation() {
      System.out.println("get articleInformation");

      List<Article> articleDetails =  Arrays.asList(

              /*
               *在这里，您可以添加数据库逻辑/流程以获取文章详细信息
               *暂时硬编码了 2 个值
               */
               new Article(100,"title1","content1"),
               new Article(101,"title2","content2")
           );

       return ResponseEntity.ok(articleDetails);
   }

   public static void main(String[] args) {
      SpringApplication.run(Example.class, args);
   }
}
```

通过如下网址测试缓存效果：

```
http://localhost:8080/get-article-info
```

RedisCacheManager 是一个由 Spring Boot 提供的 Redis 支持的 CacheManager。如果在我们的应用程序的类路径中提供了 Redis 并且所需的配置可用，则 Spring Boot 会自动配置 RedisCacheManager 实例。

Spring 提供了 spring-boot-starter-data-redis 来解决 Redis 依赖关系。Spring 为 Lettuce 和 Jedis 客户端库提供基本的自动配置。默认情况下，Spring Boot 2.0 使用 Lettuce。要获得池化连接工厂，我们需要提供 commons-pool2 依赖项。要使用 Lettuce，我们需要如下的 Gradle 依赖项。

```
dependencies {
   compile group: 'org.springframework.boot', name: 'spring-boot-starter-
data-redis', version: '2.1.3.RELEASE'
   compile group: 'org.apache.commons', name: 'commons-pool2', version: '2.6.1'
}
```

要配置 Lettuce 池，需要将 spring.redis.*前缀与 Lettuce 池连接属性一起使用。找到 Lettuce 池样本配置。

```
spring.redis.host=localhost
spring.redis.port=6379
spring.redis.password=

spring.redis.lettuce.pool.max-active=7
spring.redis.lettuce.pool.max-idle=7
spring.redis.lettuce.pool.min-idle=2
spring.redis.lettuce.pool.max-wait=-1ms
spring.redis.lettuce.shutdown-timeout=200ms
```

可以覆盖默认的 Redis 主机、端口和密码配置。如果想要无限期地阻止，可以设置 max-wait 为一个负值。

可以使用 spring.cache.*属性控制 Spring 缓存配置。

```
spring.cache.redis.cache-null-values=false
spring.cache.redis.time-to-live=600000
spring.cache.redis.use-key-prefix=true

spring.cache.type=redis
spring.cache.cache-names=articleCache,allArticlesCache
```

缓存 articleCache 和 allArticlesCache 将存活 10min。

可以创建自己的 RedisCacheManager 来完全控制 Redis 配置。需要创建 LettuceConnectionFactory bean，RedisCacheConfiguration bean 和 RedisCacheManager，如下所示。

```
@Configuration
@EnableCaching
@PropertySource("classpath:application.properties")
public class RedisConfig {
   @Autowired
   private Environment env;

   @Bean
   public LettuceConnectionFactory redisConnectionFactory() {
    RedisStandaloneConfiguration redisConf = new RedisStandaloneConfiguration();
    redisConf.setHostName(env.getProperty("spring.redis.host"));
    redisConf.setPort(Integer.parseInt(env.getProperty("spring.redis.port")));
    redisConf.setPassword(RedisPassword.of(env.getProperty("spring.redis.password")));
        return new LettuceConnectionFactory(redisConf);
   }
   @Bean
   public RedisCacheConfiguration cacheConfiguration() {
    RedisCacheConfiguration cacheConfig = RedisCacheConfiguration.defaultCacheConfig()
      .entryTtl(Duration.ofSeconds(600))
```

```
        .disableCachingNullValues();
      return cacheConfig;
      }
      @Bean
      public RedisCacheManager cacheManager() {
      RedisCacheManager rcm = RedisCacheManager.builder(redisConnectionFactory())
        .cacheDefaults(cacheConfiguration())
        .transactionAware()
        .build();
      return rcm;
      }
  }
```

这里的 RedisCacheConfiguration 是不可变类，可帮助自定义 Redis 缓存行为，如缓存到期时间，禁用缓存空值等。它还有助于自定义序列化策略。

如果要禁用缓存，则无须删除所有注释。只需在 application.properties 文件中添加以下行，SpringBoot 会为您完成一切。

```
spring.cache.type=none
```

为了在 Linux 下部署网站，可以从 Tomcat 官方网站 http://tomcat.apache.org/下载 tar.gz 包。

```
# wget http://apache.fayea.com/tomcat/tomcat-9/v9.0.16/bin/apache-tomcat-9.0.16.tar.gz
```

解压缩这个文件：

```
#tar -xf apache-tomcat-9.0.16.tar.gz
```

然后增加 Tomcat 所使用的内存。修改配置文件 catalina.sh：

```
#vi /usr/local/apache-tomcat-9.0.16/bin/catalina.sh
```

在文件 catalina.sh 的开始位置增加如下行：

```
JAVA_OPTS=-Xmx1024m
```

修改 Tomcat 配置文件 server.xml，把监听端口号从 8080 改到 80：

```
#vi /usr/local/apache-tomcat-9.0.16/conf/server.xml
```

可以把 Web 应用打一个 war 包，然后传到服务器上的 webapps/子路径下，会自动解压缩 war 包中的 Web 应用。

4.4.2 使用.NET 开发搜索界面

Elasticsearch.Net 是一个非常底层的客户端。NEST 是一个在 Elasticsearch.Net 上构建的高层.NET 客户端。这两个客户端的源代码都位于 https://github.com/elastic/elasticsearch-net。这里首先介绍使用 Elasticsearch.Net 实现搜索。

在 Visual Studio 中创建一个控制台程序。然后安装相关的 NuGet 包：

```
>Install-Package Elasticsearch.Net
```

使用底层客户端查询所有记录的代码如下：

```
//设置连接参数
var settings = new ConnectionConfiguration(
   new Uri("http://localhost:9200")).RequestTimeout(TimeSpan.FromMinutes(5));
settings.PrettyJson(); //为了调试，可以输出设置的 JSON 格式

//建立连接
var lowlevelClient = new ElasticLowLevelClient(settings);
//查询所有记录
var jsonPostData = PostData.Serializable(new {
   query = new
   {
      match all = new { }
   }
});
//返回查询结果
var searchResponse = lowlevelClient.Search<StringResponse>("news", "health",
jsonPostData);

var successful = searchResponse.Success;
var responseJson = searchResponse.Body;

Console.WriteLine(responseJson);  //输出查询结果
```

为了能够使用 ConfigurationManager 类读取配置文件，引入 ConfigurationManager 包：

```
>Install-Package System.Configuration.ConfigurationManager
```

配置文件 app.config 的示例如下：

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="Search-Uri" value="http://localhost:9200" />
  </appSettings>
</configuration>
```

配置客户端的代码如下：

```
var node = new Uri(ConfigurationManager.AppSettings["Search-Uri"]); //读配
置文件
var settings = new ConnectionConfiguration(node)
    .RequestTimeout(TimeSpan.FromMinutes(5));
```

为了使用高层客户端，首先安装 NEST 包：

```
>Install-Package NEST
```

定义对应索引结构的 POJO 类：

```
class News
{
```

```
        public string title { get; set; }      //新闻标题
        public string body { get; set; }       //新闻内容

        public override string ToString()
        {
            return string.Format("Title: '{1}', Body: '{2}'", title, body);
        }
    }
```

使用 NEST 的代码如下：

```
    var node = new Uri(ConfigurationManager.AppSettings["Search-Uri"]); //读配置文件
    var settings = new ConnectionSettings(node);  //建立连接设置
    settings.DefaultIndex("news");
    settings.DefaultTypeName("health");

    var client = new ElasticClient(settings);

    var rs = client.Search<News>(s => s.Query(q => q.MatchAll()));
    Console.WriteLine(rs.Documents);
```

使用 Newtonsoft.Json 中的 JsonConvert.SerializeObject()方法显示返回结果。首先安装 Newtonsoft.Json 包：

```
    PM> Install-Package Newtonsoft.Json
```

然后调用格式化 Json 的方法显示查询结果：

```
    String keyWords = "健康";  //查询词

    var rs = client.Search<News>(s => s
        .Index("news")
        .Type("health")
        .Size(50)
        .Query(q => q
            .Match(m => m
                .Field(f => f.title)
                .Query(keyWords)
            )));

    Console.WriteLine("结果总数 "+rs.Total);
    Console.WriteLine(JsonConvert.SerializeObject(rs.Documents));
```

如果希望使用.NET 进行 Web 开发的框架，则可以考虑选用 ASP.NET Core。ASP.NET Core 应用程序是一个控制台应用程序，嵌入了 Kestrel(https://github.com/aspnet/ AspNetCore/tree/master/src/Servers/Kestrel)Web 服务器。可在其 Program.Main 方法中创建 Web 服务器。

首先介绍在 Program.Main 方法中创建一个简单的 Echo 服务器。

打开 PowerShell 提示符并运行：

```
> mkdir echoserver
> cd echoserver
> dotnet new console
> dotnet add package Microsoft.AspNetCore -v 2.0.2
```

使用以下代码替换 Program.cs 文件的内容：

```
namespace EchoServer
{
    using Microsoft.AspNetCore.Hosting;
    using Microsoft.AspNetCore.Builder;
    using Microsoft.AspNetCore;

    class Program
    {
        static int Main(string[] args)
        {
            WebHost.CreateDefaultBuilder()
                .UseKestrel()
                .Configure(app =>
                    {
                        app.Run(httpContext =>
                        {
                            var request = httpContext.Request;
                            var response = httpContext.Response;

                            //回应标题
                            foreach (var header in request.Headers)
                            {
                                response.Headers.Add(header);
                            }

                            //回应主体
                            return request.Body.CopyToAsync(response.Body);
                        });
                    })
                .Build()
                .Run();

            return 0;
        }
    }
}
```

运行以下命令以构建应用程序：

```
>dotnet build
>dotnet run
```

Echo Server 将运行在 http://localhost:5000。测试它只是发送任何请求，服务器应该将它返回给用户。

下面尝试使用自定义标头发送 GET 请求：

```
>curl -H "Custom-Header: echo" http://localhost:5000/ -i
```

回应应该是：

```
HTTP/1.1 200 OK
Date: Tue, 19 Feb 2019 11:46:02 GMT
Server: Kestrel
Content-Length: 0
Accept: */*
Host: localhost:5000
User-Agent: curl/7.63.0
Custom-Header: echo
```

如果在 Visual Studio 中创建一个空应用，则所生成的 Program 类如下：

```
public class Program
{
   public static void Main(string[] args)
   {
      CreateWebHostBuilder(args).Build().Run();
   }

   public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
      WebHost.CreateDefaultBuilder(args)
         .UseStartup<Startup>();  //声明使用 Startup 类
}
```

Startup 类的默认实现如下：

```
public class Startup
{
   //此方法由运行时调用，使用此方法将服务添加到容器
   public void ConfigureServices(IServiceCollection services)
   {
   }

   //此方法由运行时调用，使用此方法配置 HTTP 请求管道
   public void Configure(IApplicationBuilder app, IHostingEnvironment env)
   {
      if (env.IsDevelopment())
      {
         app.UseDeveloperExceptionPage();
      }

      app.Run(async (context) =>
      {
         await context.Response.WriteAsync("Hello World!");
      });
   }
}
```

Sprinboot 在 application.properties 文件中修改 Web 服务器监听端口号，而对于 ASP.NET Core，则可以在 launchSettings.json 文件中修改 Web 服务器监听端口号。launchSettings.json

文件为一个 ASP.NET Core 应用保存特有的配置标准，用于应用的启动准备工作，包括环境变量、开发端口等。

在解决方案资源管理器中，右击项目节点，然后选择添加→新建项…选项。新建一个名为 AppSettings.json 的文件，内容如下：

```
{
  "message": "Hello, World! this message is from configuration file..."
}
```

修改 AppSettings.json 文件属性，选择“复制到输出目录”。

修改 Startup.cs 成为：

```
public class Startup
{
   public Startup()
   {
      //构建配置类的实例
      var builder = new ConfigurationBuilder()
         .AddJsonFile("AppSettings.json");
      Configuration = builder.Build();
   }

   public IConfiguration Configuration { get; set; }

   public void ConfigureServices(IServiceCollection services)
   {
   }

   public void Configure(IApplicationBuilder app, IHostingEnvironment env)
   {
      if (env.IsDevelopment())
      {
         app.UseDeveloperExceptionPage();
      }

      app.Run(async (context) =>
      {
         //从配置类读入信息
         var msg = Configuration["message"];
         await context.Response.WriteAsync(msg);
      });
   }
}
```

如果在服务器端使用模板引擎 razor，则可以增加控制类：

```
public class HomeController : Controller
{
   public IActionResult IndexAction()
   {
      return View("Index");
```

```
        }
    }
```

将 Startup.cs 文件修改成为：

```
    public class Startup
    {
        public Startup(IConfiguration configuration)
        {
            Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {
            //设置 ASP.NET Core 兼容版本
            services.AddMvc().SetCompatibilityVersion(CompatibilityVersion.Version
2 1);
        }

        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            app.UseHttpsRedirection();      //使用 HTTPS 重定向
            app.UseStaticFiles();           //使用静态文件

            app.UseMvc(routes =>
            {
               routes.MapRoute(
                   name: "default",         //路由名
                   template: "{controller=Home}/{action=IndexAction}");  //模板
            });
        }
    }
```

为了实现搜索结果翻页，可以使用 cloudscribe.Web.Pagination(https://github.com/cloudscribe/cloudscribe.Web.Pagination)。cloudscribe.Web.Pagination 是一个用于分页的 ASP.NET Core 标签助手。

如果要使用 RESTful Web API 方式构建应用，则可以修改控制类：

```
    namespace server{
       [Route("api/[controller]")]
       public class HomeController : Controller{
           [HttpGet]
           [Produces("text/html")]
           public string Get()
           {
               string responseString = @"
               <title>My report</title>
               <style type='text/css'>
               button{
                   color: green;
```

```
                }
                </style>
                <h1> Header </h1>
                <p>Hello There <button>click me</button></p>
                <p style='color:blue;'>I am blue</p>
                ";
                return responseString;
            }
        }
    }
```

为了与 Elastic 搜索实例“对话”，NEST 使用一个名为 ElasticClient 的对象。ElasticClient 负责将 C#查询转换为发送到所连接的 Elastic 集群并由其处理普通的 JSON 请求。ElasticClient 客户端是线程安全的，在整个应用程序中只有一个实例。为了实现 ElasticClient 的单例模式，可以借助 ASP.NET Core 中自带的依赖注入包 Microsoft.Extensions.DependencyInjection 实现。

首先将在 appsettings.json 文件中设置连接到 Elastic 搜索实例的配置参数。然后创建一个 ElasticConnectionSettings 类，它将用作上述设置的容器。创建一个 ElasticClientProvider 类，它将负责实例化并包含我们将使用的实际 ElasticClient。这是我们在需要 ElasticClient 时要注入的类。最后在 DI（Dependency Injection）中注册 ElasticClientProvider 类。

打开 appsettings.json 并插入以下部分。

```
"ElasticConnectionSettings": {
   "ClusterUrl": "http://localhost:9200",
   "DefaultIndex": "news"
}
```

在项目的根目录中，创建一个名为 Elastic 的文件夹。在此文件夹中，将包含所有 Elastic 特定的对象。现在使用以下实现创建一个名为 ElasticConnectionSettings 的类：

```
public class ElasticConnectionSettings
{
   public string ClusterUrl { get; set; }

   public string DefaultIndex
   {
      get
      {
         return this.defaultIndex;
      }
      set
      {
         this.defaultIndex = value.ToLower();  //小写化
      }
   }
```

```
        private string defaultIndex;
    }
```

它的属性将与 appsettings.json 文件中的属性进行匹配。DefaultIndex 属性有一个 setter，它将始终设置 appsettings.json 中指定的索引的小写值。这是因为 Elastic 中的索引必须始终为小写。

在所创建的 Elastic 文件夹中，添加一个名为 ElasticClientProvider 的新类，其中包含以下实现：

```
using Microsoft.Extensions.Options;
using Nest;

public class ElasticClientProvider
{
    public ElasticClientProvider(IOptions<ElasticConnectionSettings> settings)
    {
        //创建连接设置
        ConnectionSettings connectionSettings =
            //从 appsettings.json 获取集群 URL
            new ConnectionSettings(new System.Uri(settings.Value.ClusterUrl));
        //这将使我们能够在调试时看到发送到 Elastic 搜索的原始查询
        connectionSettings.EnableDebugMode();

        if (settings.Value.DefaultIndex != null)
        {
         //从 appsettings.json 获取索引名称
            connectionSettings.DefaultIndex(settings.Value.DefaultIndex);
        }
        //创建实际的客户端
        this.Client = new ElasticClient(connectionSettings);
    }

    public ElasticClient Client { get; }
}
```

为了使所提供的程序正常工作，必须在 Startup.cs 文件中注册一些服务。

打开 Startup.cs 并在 ConfigureServices 方法中插入以下行：

```
//从 appsettings.json 获取连接设置并将它们注入 ElasticConnectionSettings
services.AddOptions();
services.Configure<ElasticConnectionSettings>(
     Configuration.GetSection("ElasticConnectionSettings"));
```

这将告诉应用程序在 appsettings.json 中有一个名为 ElasticConnectionSettings 的部分，我们希望将其映射到之前创建的 ElasticConnectionSettings 类。

为了注册 ElasticClientProvider 类以进行注入，在 Startup.cs 文件中添加以下内容：

```
//将客户端提供程序注册为单例
```

```
services.AddSingleton(typeof(ElasticClientProvider));
```

这将使我们能够在任何需要的地方注入 ElasticClientProvider 并从中获取 ElasticClient。用法示例如下：

```
public class SomeClassThatNeedsAClient
{
    public SomeClassThatNeedsAClient(ElasticClientProvider provider)
    {
        client = provider.Client;
    }

    private readonly ElasticClient  client;
}
```

为了使用 Elasticsearch 并使用集中化的日志，可以将 Serilog 作为支持结构化日志记录的日志框架。还有用于 Elasticsearch 的 Serilog 接收器(https://github.com/serilog/serilog-sinks-elasticsearch)。

需要将 Serilog 和 Serilog.Sinks.ElasticSearch 包添加到项目中。用于装载到 Elasticsearch 的 serilog 配置的示例代码如下：

```
var logger = new LoggerConfiguration()
    .WriteTo.Elasticsearch(new ElasticsearchSinkOptions(new Uri("http://
localhost:9200"))
    {
        ModifyConnectionSettings = x => x.SetBasicAuthentication(username,
password);
    })
    .CreateLogger();
```

服务器端除了 ASP.NET Core 还可以使用 ServiceStack(https://github.com/ServiceStack/ServiceStack)。

ServiceStack 是一个简单、快速、通用且高效的全功能 Web 和 Web 服务框架，经过精心设计，旨在通过基于消息的设计减少人为复杂性并促进远程服务最佳实践，从而实现最大程度的重用。ServiceStack 利用集成的服务网关创建松散耦合的模块化服务架构。

ServiceStack Services 可通过一系列内置的快速数据格式（包括 JSON、XML、CSV、JSV、ProtoBuf、Wire 和 MsgPack）进行消费。

这里使用模板引擎 Scriban(https://github.com/lunet-io/scriban)。Scriban 是一种快速、强大、安全且轻量级的文本模板语言和.NET 引擎，具有解析 liquid(https://shopify.github.io/liquid/)模板的兼容模式。

为了使用 Scriban，首先在 Visual Studio 的程序包管理器控制台安装 Scriban 包：

```
PM> Install-Package Scriban -Version 1.2.9
```

使用 Scriban 生成文本的例子：

```
var template = Template.Parse("Hello {{name}}!");
var result = template.Render(new { Name = "Tom" }); //=> "Hello Tom!"
```

```
Console.WriteLine(result);
```

要将文本从文件加载到字符串，可以使用 StreamReader.ReadToEnd 方法。StreamReader 实现一个 TextReader，能够按给定的编码从字节流中读取字符。创建 StreamReader 的新实例，将文件路径或 Stream 作为构造函数参数传递，并指定文本文件编码（默认为 UTF-8）。

```
string text;
using (var streamReader = new StreamReader(@"c:\file.txt", Encoding.UTF8))
{
    text = streamReader.ReadToEnd();
}
```

使用 Scriban 生成网页。模板文件 searchResult.tl 的内容如下：

```
<html>
   <head>
      <title>searchresult</title>
   </head>
   <body>
      {{ content }}
   </body>
</html>
```

使用这个模板文件：

```
string text;
using (var streamReader =
    new StreamReader(@"E:\test\Scriban\searchResult.tl", Encoding.UTF8))
{
   text = streamReader.ReadToEnd();
}

var template = Template.Parse(text);
var result = template.Render(new { content="hello" });

Console.WriteLine(result);
```

输出结果如下：

```
<html>
       <head>
              <title>searchresult</title>
       </head>
       <body>
              hello
       </body>
</html>
```

4.5 检索模型

可以认为打上了和查询词同样标签的文档是相关文档。但很多时候，猜测文档是否

有相关内容是没有把握的，所以用概率来量化这种不确定性。把信息检索作为分类问题，一类是相关文档 R，还有一类是不相关的文档 NR。根据贝叶斯判别规则，如果 $P(\text{R}|\boldsymbol{D})>P(\text{NR}|\boldsymbol{D})$，则 $\boldsymbol{D}$ 是相关的文档。如果 $P(\text{R}|\boldsymbol{D})<P(\text{NR}|\boldsymbol{D})$，则 $\boldsymbol{D}$ 是不相关的文档。例如，$P(\text{R}|\boldsymbol{D})=0.8$，$P(\text{NR}|\boldsymbol{D})=0.2$，则 $\boldsymbol{D}$ 是和用户查询相关的文档。图 4-1 把“新生婴儿入户须知”这个索引库中的文档分成相关文档或者不相关文档。

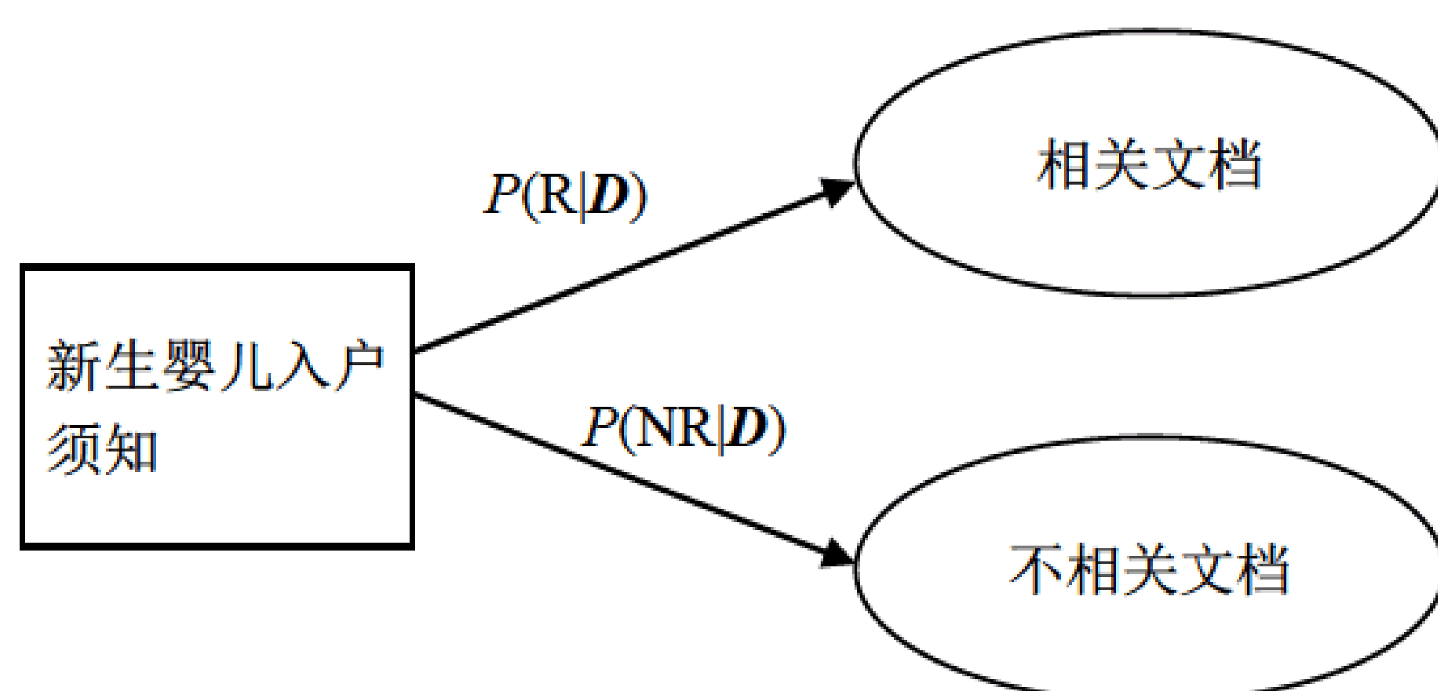


图 4-1　把信息检索看成分类问题

假设知道相关文档集合就能够计算出 $P(\boldsymbol{D}|\text{R})$。例如，如果知道某个词在相关文档集合中有多频繁出现，然后，给定一个新文档，就能直接计算出这个文档中词的组合有多大可能在相关文档集合中。

使用贝叶斯公式估计概率：

$$P(\text{R}\mid\boldsymbol{D})=\frac{P(\boldsymbol{D}\mid\text{R})P(\text{R})}{P(\boldsymbol{D})}$$

比较 $P(\text{R}|\boldsymbol{D})$和 $P(\text{NR}|\boldsymbol{D})$的值。如果满足 $P(\boldsymbol{D}|\text{R})P(\text{R})>P(\boldsymbol{D}|\text{NR})P(\text{NR})$，就把文档分到相关类。把一个文档分类成相关的条件可以写成：

$$\frac{P(\boldsymbol{D}\mid\text{R})}{P(\boldsymbol{D}\mid\text{NR})}>\frac{P(\text{NR})}{P(\text{R})}$$

左边的公式称为似然比，需要计算 $P(\boldsymbol{D}|\text{R})$和 $P(\boldsymbol{D}|\text{NR})$。为了简化计算，把文档表示成词的组合，用词概率估计 $P(\boldsymbol{D}|\text{R})$和 $P(\boldsymbol{D}|\text{NR})$。

用一个二值特征的向量表示文档的特征，表示文档中出现或者不出现某个词。把文档表示成二项特征组成的向量，$\boldsymbol{D}=(d_1,d_2,\cdots,d_t)$，如果词 i 出现在文档中，则 $d_i=1$，否则 $d_i=0$。假设词都是独立出现的，则 $P(\boldsymbol{D}|\text{R})$可以用词概率的乘积 $\prod_{i=1}^{t}P(d_i\mid\text{R})$ 计算。因为这个模型假设词独立出现，而且使用文档的二项特征，所以称为二项独立模型。

假设索引库包含 5 个词，某文档 $\boldsymbol{D}$ 根据二元假设，表示为{1,0,1,0}，其含义是这个文档出现了第 1 个、第 3 个和第 5 个词，但不包含第 2 个和第 4 个词。

用 P_i 来代表第 i 个词在相关文档集合内出现的概率，于是在已知相关文档集合的情

况下，文档 $\boldsymbol{D}$ 相关的概率为

$$P(\mathrm{R}|\boldsymbol{D})=P_1\times(1-P_2)\times P_3\times(1-P_4)\times P_5$$

式中，$1-P_2$ 代表了第 2 个词不出现在相关文档的概率，因为 $P(t_2|\mathrm{R})+P(\neg\, t_2|\mathrm{R})=1$。

为了计算 $P(\boldsymbol{D}|\mathrm{NR})$，假设用 S_i 代表第 i 个词语或单词在不相关文档集合内出现的概率，于是在已知不相关文档集合的情况下，观察到文档 $\boldsymbol{D}$ 的概率为

$$P(\boldsymbol{D}|\mathrm{NR})=S_1\times(1-S_2)\times S_3\times(1-S_4)\times S_5$$

例如，查询为："信息检索教程"，所有词项的在相关、不相关情况下的概率说明 p_i、s_i 分别如表 4-1 所示。

表 4-1 概率说明表

词　　项	信　　息	检　　索	教　　材	教　　程	课　　件
R 中的概率 p_i	0.8	0.9	0.3	0.32	0.15
NR 中的概率 s_i	0.3	0.1	0.35	0.33	0.10

假设文档 $\boldsymbol{D}_1$ 中只有 2 个词：检索课件

则：

$$P(\boldsymbol{D}|\mathrm{R})=(1-0.8)\times0.9\times(1-0.3)\times(1-0.32)\times0.15$$

$$P(\boldsymbol{D}|\mathrm{NR})=(1-0.3)\times0.1\times(1-0.35)\times(1-0.33)\times0.10$$

$$P(\boldsymbol{D}|\mathrm{R})/P(\boldsymbol{D}|\mathrm{NR})=4.216$$

返回到似然比。使用 P_i 和 S_i 得到分值：

$$\frac{P(\boldsymbol{D}\,|\,\mathrm{R})}{P(\boldsymbol{D}\,|\,\mathrm{NR})}=\prod_{i:d_i=1}\frac{p_i}{s_i}\prod_{i:d_i=0}\frac{1-p_i}{1-s_i}$$

式中，$\prod_{i:d_i=1}$ 是在文档向量中值为 1 的词对应的乘积。把上面的公式转换一下：

$$\prod_{i:d_i=1}\frac{p_i}{s_i}\prod_{i:d_i=0}\frac{1-p_i}{1-s_i}=\prod_{i:d_i=1}\frac{p_i}{s_i}\left(\prod_{i:d_i=1}\frac{1-s_i}{1-p_i}\prod_{i:d_i=1}\frac{1-p_i}{1-s_i}\right)\prod_{i:d_i=0}\frac{1-p_i}{1-s_i}$$

$$=\prod_{i:d_i=1}\frac{p_i(1-s_i)}{s_i(1-p_i)}\prod_{i}\frac{1-p_i}{1-s_i}$$

第二项在所有定义向量维度的词上运算，因此对任何文档来说，值都是一样的。对文档评分时可以忽略这项。

因为多个很小的数乘起来可能导致精度丢失或者向下溢出成为 0，所以对计算公式取对数。这样评分公式变成了：

$$\sum_{i:d_i=1} \log \frac{p_i(1-s_i)}{s_i(1-p_i)}$$

如果存在相关性反馈可以得到相关文档集合和不相关文档集合。也就是说，给定用户查询，如果可以确定哪些文档构成了相关文档集合，哪些文档构成了不相关文档集合，可以利用表 4-2 所列出的数据来估算单词概率。

表 4-2　某个查询的词出现情况的相依表

	相 关 文 档	不相关文档	文 档 总 数
d_i=1	r_i	n_i-r_i	n_i
d_i=0	$R-r_i$	$N-n_i-R+r_i$	$N-n_i$
文档总数	R	$N-R$	N

表 4-2 中第 3 行的 N 为文档集合总共包含的文档个数，R 为相关文档的个数，于是 $N-R$ 就是不相关文档集合的大小。对于某个词语或单词 d_i 来说，假设包含这个词语的文档数量共有 n_i 个，而其中相关文档有 r_i 个，那么不相关文档中包含这个单词的文档数量则为 n_i-r_i。再考虑表中第 2 列，因为相关文档个数是 R，而其中出现过单词 d_i 的有 r_i 个，那么相关文档中没有出现过这个单词的文档个数为 $R-r_i$ 个。同理，不相关文档中没有出现过这个单词的文档个数为$(N-R)-(n_i-r_i)$个。从表中可以看出，如果假设已经知道 N、R、n_i、r_i，其他参数可以靠这 4 个值推导出来。

采用最大似然估计，计算 $p_i=\frac{r_i}{R}$，$s_i=\frac{n_i-r_i}{N-R}$。为了避免 r_i=0 导致的 log0 无法计算的问题，采用相关文档和不相关文档都加 0.5 的平滑方法。这样得到 $p_i=\frac{r_i+0.5}{R+1}$，$s_i=\frac{n_i-r_i+0.5}{N-R+1}$。把这些值放入打分公式，得到

$$\sum_{i:d_i=q_i=1} \log \frac{(r_i+0.5)/(R-r_i+0.5)}{(n_i-r_i+0.5)/(N-n-R+r_i+0.5)}$$

这个打分公式没有考虑词频，相关度比考虑词频的公式低 50%。

Okapi BM25（简称 BM25）是一种相关性排序函数，适用于搜索引擎根据与给定搜索查询的相关性对匹配文档进行排序。

1. 排名函数

BM25 是一个基于单词集合的检索函数，它依据出现在每个文档中的查询词对匹配文档集合排序，而不管查询词在文档内的相互之间的联系。它不是一个单一的函数，而实际上是有略微不同的组件和参数变化的一群函数的集合。一个最典型的具体的函数

如下：

假定有一个查询词组 Q，含有关键词 $q_1,q_2,q_3,\cdots,q_n$，用 BM25 给文档 $\boldsymbol{D}$ 评分的公式为

$$\mathrm{SCORE}(\boldsymbol{D},Q)=\sum_{i=1}^{n}\mathrm{IDF}(q_i)\cdot\frac{f(q_i,\boldsymbol{D})(k_1+1)}{f(q_i,\boldsymbol{D})+k_1(1-b+b\dfrac{|\boldsymbol{D}|}{\mathrm{avg}\ d_1})}$$

式中，$f(q_i,\boldsymbol{D})$是检索词 q_i 在文档 $\boldsymbol{D}$ 中的频率；$|\boldsymbol{D}|$是文档 $\boldsymbol{D}$ 以单词为单位的长度；avg d_1 是从抽取出的文档的文本集合的平均文档长度；k_1 和 b 是自由参数，通常选择 $k_1=2.0$ 和 $b=0.75$。$\mathrm{IDF}(q_i)$是检索词 q_i 的 IDF（文档频率倒数）权重。$\mathrm{IDF}(q_i)$的通常计算公式为

$$\mathrm{IDF}(q_i)=\log\frac{N-n(q_i)+0.5}{n(q_i)+0.5}$$

其中，N 是集合中文档的总数；$n(q_i)$是包含 q_i 的文档个数。

4.5.1 使用 BM25 检索模型

基本上可以在索引设置中定义类似于自定义分析器的自定义 BM25 相似度，也就是定义 b 和 k_1 的值。例子如下：

```
# curl -XPUT "http://<server>/<index>" -d '
{
  "settings": {
    "similarity": {
      "custom_bm25": {
        "type": "BM25",         //相似性类型为 BM25
        "b":    0 ,             //设定 b 的值
         "k1" : 0.9             //设定 k1 的值
      }
    }
  }'
```

4.5.2 参数调优

为了在 BM25 中调整这些参数，它对数据集非常依赖。简单的方法是：调整参数，然后检查结果，如果不满意，更改参数并再次测试结果。也可以使用像遗传算法或蚁群算法（ant colony optimization）这样的启发式算法自动调参。

TensorFlow 归一化调优的经典方法是旋转归一化方法。但这个方法存在集合依赖问题。

可以下载 TREC 提供的数据集进行自动调参。数据集中的特殊搜索（ad hoc search）

对一个固定的文档集合，根据用户输入的查询问题返回相关性降序输出的文档列表。

Jenetics(https://github.com/jenetics/jenetics)是一个用 Java 语言编写的遗传算法库。它被设计成明确地分离算法的几个概念，例如基因、染色体、基因型、表型、群体和适应度函数（fitness function）。

在一些包含时间信息的索引中，在许多情况下，期望的结果是保持相关性得分，并为更近期的匹配提供额外的提升（更高的分数），因为数据更新鲜。为了实现这一点，可以使用 Elasticsearch 的函数评分。

```
    final MultiMatchQueryBuilder multiMatchQuery =
           QueryBuilders.multiMatchQuery("Bababooey",
                    "title^0.8", "url^0.6", "description^0.3")
              .type(MultiMatchQueryBuilder.Type.BEST_FIELDS);

    final FunctionScoreQueryBuilder functionScoreQuery =
      QueryBuilders.functionScoreQuery(multiMatchQuery);
    functionScoreQuery.scoreMode("multiply");
    functionScoreQuery.boostMode(CombineFunction.MULT);
    functionScoreQuery.add(ScoreFunctionBuilders.gaussDecayFunction("postDate
","130w")
           .setOffset("26w").setDecay(0.3));
```

这个例子提升了过去半年发布的所有文档。超过 6 个月的文档将逐渐减少，直到达到两年半的门槛。超过两年半的文档将不会根据新近度获得任何额外的评分。通过更改 setOffet 和 setDecay，可以将此提升更改至两周或任何可能的提升窗口。

4.6　搜索中文优化

在 Elasticsearch 中，可以插件方式提供自定义的文本分析功能。这里介绍如何创建一个支持中文分词的插件。选择使用 Gradle 构建中文分词插件。build.gradle 文件内容如下：

```
    apply plugin: 'java-library'
    apply plugin:'java'
    apply plugin: 'eclipse'

    repositories {
       maven {url 'http://maven.aliyun.com/nexus/content/groups/public/'}  //
存储库
       mavenCentral()
    }

    tasks.withType(JavaCompile) {
       options.encoding = 'UTF-8'    //指定编码集
```

```
    }

    ext {
        luceneVersion= "7.5.0"          //指定 Lucene 版本号
        randomizedrunnerVersion = "2.6.0"
    }

    jar {
        baseName = 'seg'                //指定 jar 包的名字
        version = '1.0'                 //指定版本号
    }

    dependencies {                      //指定依赖的包
        compile group: 'org.elasticsearch', name: 'elasticsearch', version: '6.5.4'
        compile "org.slf4j:slf4j-api:1.5.11"
        compile "org.slf4j:slf4j-simple:1.5.11"

        testCompile "org.apache.lucene:lucene-test-framework:${luceneVersion}"
        testCompile
"com.carrotsearch.randomizedtesting:randomizedtesting-runner:${randomizedrunn
erVersion}"
        testCompile group: 'org.elasticsearch.test', name: 'framework', version:
'6.5.4'
        testCompile group: 'junit', name: 'junit', version: '4.11'
    }
```

采用多模式匹配思想实现的最大长度匹配算法每次从词典中找和待匹配串前缀最长匹配的词，如果找到匹配词，则把这个词作为切分词，待匹配串减去该词；如果词典中没有词匹配上，则按单字切分。

多模式三叉检索树匹配方法实现的 CnTokenizer 代码如下。

```
    public class CnTokenizer extends Tokenizer {
       //根据词表初始化三叉 Trie 树
       private  static  TernarySearchTrieC  dic  =  new  TernarySearchTrieC
("WordList.txt");
       private CharTermAttribute termAtt;                      //词属性
       private OffsetAttribute offsetAtt;                      //位置属性
       private static final int IO_BUFFER_SIZE = 4096;         //IO 缓存区大小
       private char[] ioBuffer = new char[IO_BUFFER_SIZE];     //IO 缓存区

       private boolean done;
       private int i = 0;              //用来控制匹配的起始位置的变量
       private int upto = 0;

       public CnTokenizer(AttributeFactory factory) {
          super(factory);
          this.termAtt = addAttribute(CharTermAttribute.class);
          this.offsetAtt = addAttribute(OffsetAttribute.class);
          this.done = false;
```

```
    }

    public void resizeIOBuffer(int newSize) {
        if (ioBuffer.length < newSize) {
            //不够大。创建一个新的数组，多分配内存并保留内容
            final char[] newCharBuffer = new char[newSize];
            System.arraycopy(ioBuffer, 0, newCharBuffer, 0, ioBuffer.length);
            ioBuffer = newCharBuffer;
        }
    }

    @Override
    public boolean incrementToken() throws IOException {
        if (!done) {
            clearAttributes();
            done = true;
            upto = 0;
            i = 0;
            while (true) {
                final int length = input.read(ioBuffer, upto, ioBuffer.length
                        - upto);
                if (length == -1)
                    break;
                upto += length;
                if (upto == ioBuffer.length)
                    resizeIOBuffer(upto * 2);
            }
        }

        if (i < upto) {
            char[] word = dic.matchLong(ioBuffer, i, upto);//正向最大长度匹配
            termAtt.setEmpty();
            if (word != null)//已经匹配上
            {
                termAtt.copyBuffer(word, 0, word.length);
                offsetAtt.setOffset(i, i +word.length);
                i += word.length;
            } else {
                termAtt.copyBuffer(ioBuffer, i, 1);
                offsetAtt.setOffset(i, i+1);
                ++i;        //下次匹配点在这个字符之后
            }
            return true;
        }

        return false;
    }
}
```

为了能在 Lucene 中连续执行，CnTokenizer 需要重写 reset()方法。

```
@Override
```

```
    public void reset() throws IOException {
      super.reset();
      this.i=0;
      this.done = false;
      this.upto = 0;
    }

    @Override
    public void end() throws IOException {
      //设置最后的偏移量
      final int finalOffset = correctOffset(upto);
      offsetAtt.setOffset(finalOffset, finalOffset);
    }
```

测试 CnTokenizer：

```
    String text = "中国成立了";
    StringReader input = new StringReader(text);

    //创建 AttributeImpl 实例的 AttributeFactory
    AttributeFactory factory = AttributeFactory.DEFAULT ATTRIBUTE FACTORY;
    CnTokenizer tokenizer = new CnTokenizer(factory);
    tokenizer.setReader(input);
    //将 tokenizer 重置到开头
    tokenizer.reset();

    //从 TokenStream 获取 CharTermAttribute
    CharTermAttribute termAtt = tokenizer.addAttribute(CharTermAttribute.class);
    //从 TokenStream 获取 OffsetAttribute
    OffsetAttribute offsetAtt = tokenizer.addAttribute(OffsetAttribute.class);

    while (tokenizer.incrementToken()) {
       System.out.println(termAtt.toString() + " /" +  //输出切分出来的词
                     //输出切分出来的词的位置信息
                     offsetAtt.startOffset() + "," + offsetAtt.endOffset());
    }
    tokenizer.close();
```

支持中文的 AnalyzerProvider 类：

```
    public class CnAnalyzerProvider extends AbstractIndexAnalyzerProvider
<NgramAnalyzer>{
       private static final Logger logger = LoggerFactory.getLogger(CnAnalyzerProvider.
class);
       private final NgramAnalyzer analyzer;

       @Inject
       public CnAnalyzerProvider(IndexSettings indexSettings, Environment env,
@Assisted String name, @Assisted Settings settings){
          super(indexSettings, name, settings);
          Path pluginDir = env.pluginsFile();  //得到插件目录
          //词典文件放在插件目录的子目录下
```

```
        String dicPath=new File(pluginDir.toFile(),"seg/dic").getPath();
        logger.info("plugin dic Dir:"+dicPath);
        analyzer = new NgramAnalyzer(dicPath+"/");
    }

    @Override
    public NgramAnalyzer get() {
        return analyzer;
    }
}
```

测试 CnAnalyzerProvider：

```
final String text = "保护用户隐私";
Settings settings = Settings.builder().put(IndexMetaData.SETTING_VERSION_CREATED,
                                Version.CURRENT)
  .put(IndexMetaData.SETTING_NUMBER_OF_REPLICAS, 0)
  .put(IndexMetaData.SETTING_NUMBER_OF_SHARDS, 1)
      .put(IndexMetaData.SETTING_INDEX_UUID, UUIDs.randomBase64UUID())
      .put(Environment.PATH_HOME_SETTING.getKey(), "").build();
Environment environment = TestEnvironment.newEnvironment(settings);
IndexMetaData metaData = IndexMetaData.builder(IndexMetaData.INDEX_UUID_NA
VALUE). settings(settings). build();
IndexSettings indexSettings = new IndexSettings(metaData, Settings.EMPTY);

CnAnalyzerProvider p =
  new CnAnalyzerProvider(indexSettings, environment, null, Settings.EMPTY);
NgramAnalyzer analyzer = p.get();

TokenStream stream = analyzer.tokenStream("field", new StringReader(text));

//从 TokenStream 获取 TermAttribute
CharTermAttribute termAtt = stream.addAttribute(CharTermAttribute.class);
OffsetAttribute offsetAtt = stream.addAttribute(OffsetAttribute.class);
TypeAttribute typeAtt = stream.addAttribute(TypeAttribute.class);

stream.reset();

//打印所有符号，直到流耗尽
while (stream.incrementToken()) {
   System.out.println(termAtt.toString() + " /" + offsetAtt.startOffset() + ","
       + offsetAtt.endOffset() + " " + typeAtt.type());
}

stream.end();
stream.close();
```

plugin 脚本用于安装、列出和删除插件。它默认位于$ES_HOME/bin 目录。列出插件：

```
# /usr/share/elasticsearch/bin/elasticsearch-plugin list
```

利用 install 参数安装插件，例如安装日文插件：

```
# /usr/share/elasticsearch/bin/elasticsearch-plugin  install analysis-kuromoji
```

4.7 Elasticsearch 源代码分析

Elasticsearch 使用 Lucene 实现全文搜索功能。Elasticsearch 通过模块来分解功能实现。Elasticsearch 中的模块是 Guice 模块组件，它提供配置信息并绑定 Elasticsearch 的各种接口的特定实现。

本节首先介绍 Guice 框架以及 Elasticsearch 所使用的网络通信框架 Netty，然后介绍 Elasticsearch 使用的 Lucene 源代码。

4.7.1 导入源代码到 Eclipse

首先确认已经将 Eclipse 的默认编码格式修改为 UTF-8。如果还没有修改过，则可以从菜单栏的 Window 选项下修改，然后确认使用的 JDK 的版本为 10 以上。

在构建发布包之前，'check'运行所有的测试作为一项规则，这是必须调用的东西。只应该有很少的属性来改变构建行为，并且它们需要保留用于高级用途，如调试等。

我们的'check'规则有一些例外。向后可比性（backwards comparability，BWC）测试就是其中之一。这些测试声称可以保留向后的可比性，它们执行从每个兼容版本升级到当前版本的事情，这可能需要花费大量时间才能在开发周期中有很多这样的版本，所以不要将所有的 BWC 测试作为检查的一部分，而是为它们提供专门的持续集成工作。

4.7.2 Guice 框架

Guice 是 Google 公司开发的一个开源依赖注入框架（IOC）。Guice 减少了对工厂类的需求以及在 Java 代码中使用 new 关键词。可以将 Guice 的@Inject 注解视为新的 new。使用 Guice 能让代码更容易更改，进行单元测试和在其他上下文中重用。

Elasticsearch 在启动时，基于其配置文件和运行时环境来搜集不同模块，并创建一个 Injector。简单说，Injector 就是一个不需要提供构建参数即可构建类的实例的对象。Injector 将会使用它的配置完的模块来定位所有请求的依赖，并以一种拓扑顺序来创建出这些实例。这样的做法可节约大量时间，并帮助我们创建出一个可复合的模块系统。

Elasticsearch 服务启动时通过 ModuleBuilder 类来进行模块注入。ModuleBuilder 是 Guice 的封装。

Elasticsearch 源代码中集成了 Guice。相关代码位于 org.elasticsearch.common.inject 包中。Guice 会扫描 inject 注释，并对方法中出现的参数实例寻找对应注册的实例进行初始化。

Elasticsearch 中的模块是一个 Guice 模块部件完成配置信息和绑定 Elasticsearch 各类接口的特定实现。

Guice 的 Provider 类可以返回特定类型的对象。Elasticsearch 使用 Provider 创建和返回 Analyzer 对象。

使用 Provider 的一个例子，首先定义一个接口：

```
public interface MyInterface {
   String foobar();
}
```

然后是接口的实现类 MyClass：

```
public class MyClass implements MyInterface {
      private String providerName;   //记录提供者的名字

       public MyClass(String providerName) {
        this.providerName = providerName;
       }

      @Override
       public String foobar() {
         return String.format("Hi! I am [%s], " + "and I was instantiated using
[%s]", getClass().getSimpleName(), providerName);   //返回调用信息
      }

}
```

Provider 的子类 MyInterfaceProvider 提供 MyClass 的实例（instance）：

```
import com.google.inject.Provider;

public class MyInterfaceProvider implements Provider<MyClass>{//实现提供者接口

      @Override
      public MyClass get() {
       return new MyClass(getClass().getSimpleName());   //用类名实例化 MyClass
      }

}
```

Module 的子类建立绑定：

```
import com.google.inject.AbstractModule;

public class MyModule extends AbstractModule {

   @Override
```

```
    protected void configure() {
        //绑定 MyInterface 接口到 Provider 子类
        bind(MyInterface.class).toProvider(MyInterfaceProvider.class);
    }

}
```

使用 Guice 得到 MyInterface 的对象：

```
import com.google.inject.Guice;
import com.google.inject.Injector;

public class ProviderSample {

    public static void main(String[] args) {
        Injector injector = Guice.createInjector(new MyModule());
                                        //创建注入器
        MyInterface myObject = injector.getInstance(MyInterface.class);
                                        //通过注入器得到实例

        System.out.println(myObject.foobar());  //输出调用信息
    }
}
```

运行 ProviderSample 输出：

```
Hi! I am [MyClass], and I was instantiated using [MyInterfaceProvider]
```

4.7.3 Netty 异步 IO 框架

Netty 是一个 NIO 客户端服务器框架，可以使用 Netty 快速轻松地开发协议服务器和客户端等网络应用程序。Netty 极大地简化了 TCP 和 UDP 套接字服务器等网络编程。Elasticsearch 采用 Netty 实现 HTTP 异步通信协议。

Elasticsearch 中的服务器端 Netty4HttpServerTransport 位于 transport-netty4 模块。客户端的 PreBuiltTransportClient 也依赖 transport-netty4 模块。

Netty 是 Reactor 设计模式的一个实现。Reactor 设计模式是用于处理由一个或多个输入同时发送到服务处理程序的服务请求的事件处理模式。然后，服务处理器多路复用输入的请求并将其同步分派到相关联的请求处理程序。

例如，对于一个具有 1 万个持久连接的服务器，这 1 万个连接将共享工作线程。一个工作线程将处理多个连接/通道。这就是为什么不阻止工作线程非常重要的原因。

io.netty.bootstrap.ServerBootstrap 负责引导服务器启动 NIO 服务。使用 ServerBootStrap 的代码如下：

```
int workerCount=1;

ServerBootstrap serverBootstrap = new ServerBootstrap();
```

```
ThreadFactory f = new DefaultThreadFactory("thread pool"); //守护线程工厂
//Reactor 单线程模型
//IO 事件作为一个触发器，网络请求事件在 NioEventLoop 中进行处理
serverBootstrap.group(new NioEventLoopGroup(workerCount, f));
```

4.7.4 分布式设计与实现

Elasticsearch 使用主节点管理集群。主节点是集群中唯一可以更改集群状态的节点。这意味着如果主节点重新启动或关闭，那么将无法对群集进行任何更改。任何一个时刻集群中只能有一个主节点。但是为了避免单点失败，需要有多个候选主节点。

包括主节点在内的每个节点都知道每个文档所在的位置，并可将搜索请求直接转发到保存了我们感兴趣的数据的节点。当搜索请求发送到一个节点时，该节点就成为协调节点。这个节点的工作是将搜索请求广播给所有涉及的分片，并将它们的响应收集到全局排序的结果集中，以便返回给客户端。

首次启动 Elasticsearch 集群需要在集群中的一个或多个符合主节点的节点上显式定义初始的符合主节点的节点集。这称为群集自举。这仅在群集首次启动时才需要：已加入群集的节点将此信息存储在其数据文件夹中，而加入现有群集的新启动节点将从群集的选定主节点获取此信息。

Elasticsearch 集群默认使用 Zen Discovery（Zen 发现机制）管理。

Zen 发现机制是 Elasticsearch 默认的内建模块。它提供了多播和单播两种发现方式，能够很容易地扩展至云环境。

Zen 发现机制是和其他模块集成的，例如所有节点间的通信必须用 Transport 模块来完成。Transport 这层是自己可以扩展的，thrift 也是一个 Transport 模块。

Elasticsearch 运行时会启动两个探测进程。一个进程用于从主节点向集群中的其他节点发送 ping 请求来检测节点是否正常可用。另一个进程的工作反过来，由其他的节点向主节点发送 ping 请求来验证主节点是否正常且忠于职守。

一个集群有一个唯一的名字，包含一个或者多个节点。集群会在所有的节点中自动选择一个作为主节点，如果主节点宕机了，则会自动选择另外一个节点作为主节点。一个经典的主节点选举算法是同行评审出版算法（peer-reviewed published algorithm）。

Elasticsearch 采用了一个简单的方法选出主节点：它根据编号来选择节点，较小的编号更有可能成为主节点。DiscoveryNode 类中记录了节点编号。选举算法的实现代码在 ElectMasterService.electMaster()方法中。

为了避免一个集群中存在不同的主节点，也就是避免脑裂，需要合理地设置 elasticsearch.yml 配置文件。

假设可以成为集群一部分的 ES 节点的数量（ES 进程而不是物理机器的数量）是 N，那么在一个有 $N>2$ 个节点的集群上，可以设置 discovery.zen.minimum_master_nodes 的值不小于$(N/2)+1$。

理想的拓扑结构是集群有 3 个专用的候选主节点（即 master: true 并且 data:false），并且 discovery.zen.minimum_master_nodes 设置为 2。这样无论集群中有多少数据节点，都不需要改变节点设置。

例如，候选主节点的配置如下：

```
node.master: true
node.data: false
discovery.zen.minimum_master_nodes: 2
```

数据节点的配置如下：

```
node.master: false
node.data: true
discovery.zen.minimum_master_nodes: 2
```

每个文档都保存在单独的主分片里。当对一个文档做索引的时候，首先对主分片做索引，然后在所有主分片的副本里做索引。默认一个索引有 5 个主分片，可以调整主分片的数量以控制一个索引中容纳文档的数量。索引创建之后，不可以更改主分片数。即使只在一台机器上安装 ES，也可能会有 5 个独立的索引库。

每个主分片可以有 0 个或者多个副本。副本是主分片的复制品，有两个作用：

- 提高容错能力：如果主分片宕机，副本分片可以被提升至主分片。
- 提高性能：搜索访问可以分布在主分片和副本分片之间。

默认每个主分片有一个副本分片，但副本分片数量可以在已经存在的索引上动态调整。在同一个节点上，副本分片不会被当作主分片启动。

用三个节点的集群举例说明索引分片的用处：第一台机器中存放索引分片 a、b、c，第二台机器中存放索引分片 a、b、d，第三台机器中存放索引分片 b、c、d。这样实现了提升索引整体容量的同时，也提升了性能和容错能力。

新增一个节点，Elasticsearch 会自动把索引数据同步到这个新增的节点上。控制界面中显示的紫色的块表示正在迁移这部分数据。

主控节点管理 shard（分片）的分配。当新机器进来或者有旧机器失效的时候，就会重新分配 shard。

4.7.5 使用 Lucene

Lucene 完成基本的搜索功能只有一个不依赖外部程序包的一个.jar 文件。因为这个文件是一个核心文件，所以这个文件称为 lucene-core-Version.jar。例如 Lucene 的 7.6.0

版本称为 lucene-core-7.6.0.jar。可以在/usr/share/elasticsearch/lib 路径找到这个 jar 包。

待查询的文档集合按词组织成倒排索引。Lucene 中的索引库是位于一个目录下的一些二进制文件。Lucene 中的索引库称为 Index。和一般的数据库不一样，Lucene 不支持定义主键。在 Lucene 中并不存在一个称为 Index 的类。Lucene 通过 IndexWriter 来写索引，通过 IndexReader 读索引。索引库在物理形式上一般位于一个路径下的一系列文件。

先介绍如何创建索引库，然后介绍如何搜索索引库。总的来说，往 Lucene 中放的是文档，查询的是词，查询返回的也是文档。使用 Lucene 实现搜索的基本概念如图 4-2 所示。

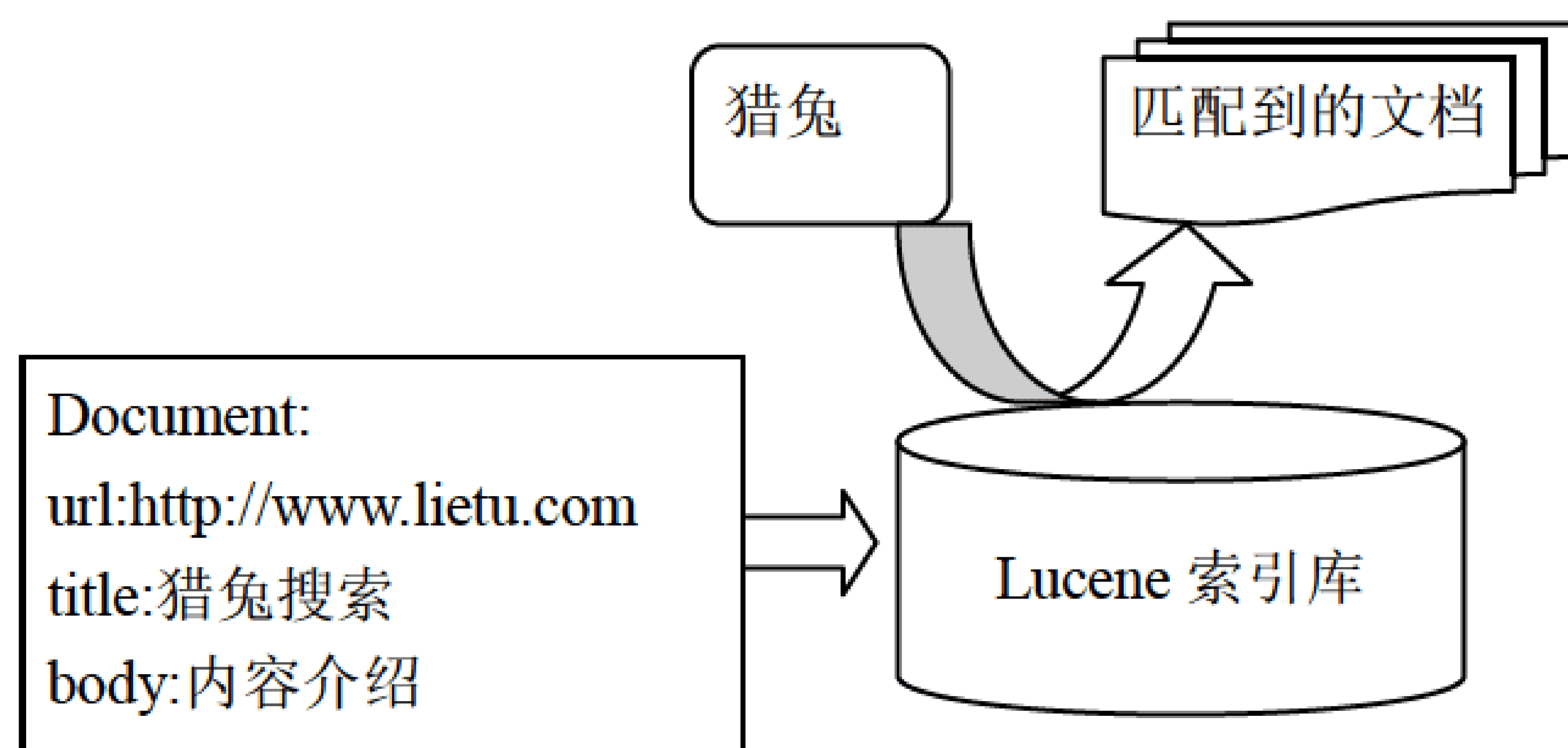


图 4-2　Lucene 搜索的基本概念

在 Eclipse 中创建一个 Java 控制台项目。创建 lib 目录，然后把 lucene-core-7.6.0.jar 文件复制到 lib 目录下。在项目属性中增加对 lucene-core-7.6.0.jar 文件的引用。

首先通过一个在内存中建立索引的例子来熟悉 Lucene 的 API：

```
public class TestBasicBooleanQuery {
   private static final String FIELD = "contents"; //查询列的名字

   public static void main(String[] args) throws Exception {
      //设置 Lucene 使用内存索引
      Directory directory = new ByteBuffersDirectory();
      //创建写索引需要用到的文本分析器
      Analyzer analyzer = new StandardAnalyzer();
      IndexWriterConfig iwc = new IndexWriterConfig(analyzer);
      IndexWriter writer = new IndexWriter(directory, iwc);  //IndexWriter
写索引

      //索引一些文档
      writer.addDocument(createDocument("1", "foo bar baz"));
      writer.addDocument(createDocument("2", "red green blue"));
      writer.addDocument(createDocument("3",
            "The Lucene was made by Doug Cutting"));
      writer.close();
```

```
        IndexReader reader = DirectoryReader.open(directory);

        IndexSearcher searcher = new IndexSearcher(reader);

        Term t1 = new Term(FIELD, "lucene");
        TermQuery q1 = new TermQuery(t1);  //使用 TermQuery 查询基本词

        Term t2 = new Term(FIELD, "doug");
        TermQuery q2 = new TermQuery(t2);

        //合取查询
        BooleanQuery.Builder booleanQuery = new BooleanQuery.Builder();
        booleanQuery.add(q1,BooleanClause.Occur.MUST); //必须包含这个条件
        booleanQuery.add(q2,BooleanClause.Occur.MUST);

        //显示搜索结果
        TopDocs topDocs = searcher.search(booleanQuery.build(), 10);
        //最多返回 10 条结果
        for (ScoreDoc scoreDoc : topDocs.scoreDocs) {
          Document doc = searcher.doc(scoreDoc.doc);
          System.out.println(doc);
        }
    }

    private static Document createDocument(String id, String content) {
        //创建文档
        Document doc = new Document();
        //向文档中增加一个字符串类型的索引列
        doc.add(new Field("id", id, StringField.TYPE STORED));
        //向文档中增加一个字符串类型的索引列
        doc.add(new Field("contents", content, TextField.TYPE STORED));
        return doc;
    }

}
```

Elasticsearch 实现了可靠的异步写入以实现长期持久性。Elasticsearch 事务日志确保可以安全地将数据索引到 Elasticsearch 中，而无需为每个文档执行低级 Lucene 提交。提交 Lucene 索引会在 Lucene 级别创建一个新段，会导致大量硬盘 I/O 影响性能。

为了接收索引文档并使其可搜索而不需要完整的 Lucene 提交，Elasticsearch 将其添加到 Lucene IndexWriter 并将其附加到事务日志中。在每个 refresh_interval 之后，它将在 Lucene 索引上调用 reopen()，这将使数据可以在不需要提交的情况下进行搜索。这是 Lucene 近实时搜索 API 的一部分。当 IndexWriter 最终由于事务日志的自动刷新或由于显式刷新操作而提交时，先前的事务日志将被丢弃并且新的事务日志将取代它。

如果需要恢复，将首先恢复写入 Lucene 硬盘的段，然后重放事务日志，以防止丢

失尚未完全提交到硬盘的操作。

可以使用 Luke(https://github.com/DmitryKey/luke)检查索引内容。

4.8　本章小结

ElasticLowLevelClient 是一个低级别，无依赖关系的客户端，对于如何构建和表示用户请求和响应没有任何意见。

搜索应用的 RESTful 客户端可以使用 XAML 应用程序。可以使用 ReactiveUI(https://github.com/reactiveui/ReactiveUI)实现下拉搜索提示词列表显示功能。

除了 Elasticsearch，还可以使用 Solr 搭建分布式搜索集群。

第 5 章 分布式计算平台

共识是让集群中的所有进程根据每个进程的投票就某些特定值达成一致的任务。所有进程必须在相同的值上达成一致，并且这个值必须是由至少一个进程提交的值。在最基本的情况下，值可以是二进制（0 或 1），所有的进程可以使用这个值来决定是否执行某些操作。

5.1 Atomix 框架

Atomix 3.0 是一个用于构建容错分布式系统的全功能框架。它提供了构建可伸缩和容错分布式系统通常所需的一组高级原语。

Atomix 具有可靠的数据一致性保证，Atomix 实现了建立在 Raft 共识协议的实现之上的分布式协调原语，即使在发生机器或网络故障时也能始终保证。

5.1.1 Raft 协议

Raft 协议提供了一种在计算系统集群中分布状态机的通用方法，确保集群中的每个节点都同意一系列相同的状态转换。Raft 以 Reliable、Replicated、Redundant 和 Fault-Tolerant 命名。

Raft 协议通过领导方法实现共识。该集群只有一个当选的领导者，负责管理集群其

他服务器上的日志复制。这意味着领导者可以决定新条目的放置以及在它与其他服务器之间建立数据流而无需咨询其他服务器。现有领导者一直领导，直到失败或断开连接，在这种情况下，新的领导者当选。

共识问题在 Raft 中被分解为下面列出的两个相对独立的子问题。

1. 领导人选举

当现有领导者失败或启动算法时，需要选出新的领导者。

在这种情况下，新的时期在群集中开始。时期是服务器上的任意时间段，在此期间需要选择新的领导者。

每个时期都以领导者选举开始。如果选举成功完成（即选出一名领导人），则将继续由新领导人正常运作。如果选举失败，新的时期就会开始新一轮选举。

领导者选举由候选服务器启动。如果服务器在称为选举超时的时间段内没有收到领导者的通信，则服务器成为候选者，因此这台服务器假定不再有代理领导者。这台服务器通过增加时期计数器，投票给自己作为新的领导者，并通过向所有其他服务器发送消息来请求投票以开始选举。服务器每个时期只会投票一次，先到先得。如果候选人收到来自另一台服务器的消息，其期限号码至少与候选人当前的期限一样大，则候选人的选举将被取消，候选人将变为追随者并将该领导者视为合法。如果候选人获得多数选票，那么它就会成为新的领导者。如果两者都没有发生，例如，由于分裂投票，那么新的时期开始，新的选举开始。

Raft 使用随机选举超时来确保快速解决分割投票问题。这应该可减少分裂投票的机会，因为服务器不会同时成为候选者：单个服务器将超时，赢得选举，然后成为领导者并在任何关注者成为候选者之前向其他服务器发送心跳消息。

2. 日志复制

领导者负责日志复制的管理，并接收客户端请求。每个客户端请求都包含一个由集群中复制的状态机执行的命令。在作为新条目附加到领导者的日志之后，每个请求将作为 AppendEntries 消息转发给关注者。如果关注者不可用，则领导者将无限期地重试 AppendEntries 消息，直到所有关注者最终存储日志条目。

一旦领导者收到其大多数关注者的确认，该条目已被复制，领导者将该条目应用于其本地状态机，把该请求视为已提交。此事件还会提交领导者日志中的所有先前条目。一旦关注者得知提交了日志条目，关注者就会将该条目应用于其本地状态机。这样，贯穿了集群为所有服务器之间的日志提供一致性、确保遵守日志匹配的安全规则。

在领导者崩溃的情况下，日志可能会不一致，而旧的领导者的某些日志未通过群集完全复制。新的领导者将通过强制关注者复制其自己的日志来处理不一致。为此，对于

每个关注者，领导者将其日志与来自跟随者的日志进行比较，找到他们同意的最后一个条目，然后删除跟随者日志中此关键条目之后的所有条目，并将这个条目替换为自己的日志条目。此机制将恢复出现故障的群集中的日志一致性。

5.1.2　使用 Atomix

Atomix 打包在一个模块层次结构中，允许用户仅依赖于自己打算使用的那些功能。几乎所有用户都希望使用 Atomix 核心模块。该模块由 Atomix 工件 ID 标识：

```
<dependencies>
  <dependency>
    <groupId>io.atomix</groupId>
    <artifactId>atomix</artifactId>
    <version>3.0.6</version>
  </dependency>
</dependencies>
```

另外，大多数集群依赖于用于复制分布式基元的一组协议。所需的依赖项取决于系统的一致性、容错性和持久性要求。不同的用例可能需要不同的依赖关系。可以使用 atomix-raft 复制协议，这样依赖项就成为：

```
<dependencies>
  <dependency>
    <groupId>io.atomix</groupId>
    <artifactId>atomix</artifactId>
    <version>3.0.6</version>
  </dependency>
  <dependency>
    <groupId>io.atomix</groupId>
    <artifactId>atomix-raft</artifactId>
    <version>3.0.6</version>
  </dependency>
</dependencies>
```

使用 Atomix 的第一步是形成一个集群。要形成集群，通常需要引导一组节点。此外，如果使用分布式原语，则必须配置一个或多个分区组。

用户可以使用各种方法在 Atomix 集群上运行。这些方法中最基本的是使用 Java API，它提供的主要性能是一致性和灵活性。

Atomix 中的核心 Java API 是 Atomix 对象。Atomix 在很大程度上依赖于构建器模式来构造用于通信和协调分布式系统的高级对象。

要创建新的 Atomix 实例，应先创建 Atomix 构建器：

```
AtomixBuilder builder = Atomix.builder();
```

应使用本地节点来配置构建器：

```
builder.withId("member1")
```

```
    .withAddress("10.192.19.181")
    .build();
```

除了配置本地节点信息之外，还必须为每个实例配置发现配置，以用于发现群集中的其他节点。最简单的发现形式是 BootstrapDiscoveryProvider：

```
builder.withMembershipProvider(BootstrapDiscoveryProvider.builder()
  .withNodes(
    Node.builder()
      .withId("member1")
      .withAddress("10.192.19.181")
      .build(),
    Node.builder()
      .withId("member2")
      .withAddress("10.192.19.182")
      .build(),
    Node.builder()
      .withId("member3")
      .withAddress("10.192.19.183")
      .build())
  .build());
```

最后，必须使用一个或多个分区组配置实例。可以使用配置文件配置公共分区组。

```
builder.addProfiles(Profile.dataGrid());
```

通常，需要强一致性保证的集群配置有 CORE 节点和至少一个 RaftPartitionGroup，并且为 DATA 性能和可伸缩性而设计的集群使用 PrimaryBackupPartitionGroup。

```
Atomix atomix = builder.build();
```

在实例上调用 start()方法以启动节点：

```
atomix.start().join();
```

start()方法返回一个节点加入集群后就会完成 future 实例。

为了形成由 CORE 节点和 Raft 分区组组成的集群，必须同时启动大多数实例以允许 Raft 分区形成仲裁。start()方法返回的 future 会直到所有分区都能够形成才会完成。

如果 Atomix 实例在启动时无限期阻塞，请确保启用 DEBUG 日志记录以调试该问题。

有多种方法可以管理 Atomix 集群并与之交互。Atomix 代理是 Java API 的便捷替代方案。代理程序是一个独立的 Atomix 节点，其行为就像 Java 节点一样，但代之以公开 REST API 以进行客户端交互。代理可用于在客户/服务器体系结构中配置 Atomix 群集或提供对 Atomix 原语的多语言访问。

要使用 Atomix 代理，首先应使用 Maven 下载并构建 Atomix：

```
>mvn clean package
```

构建项目后，要运行代理程序，必须设置$ATOMIX_ROOT 环境变量：

```
export ATOMIX_ROOT=./
```

代理程序使用 bin/atomix-agent 脚本运行：

```
>bin/atomix-agent -h
```

使用-h 选项可查看代理脚本的选项列表。

5.2 gRPC 框架

gRPC 是一个现代的、开源的、高性能的远程过程调用（RPC）框架，可以在任何地方运行。gRPC 使客户端和服务器应用程序能够透明地进行通信，并简化了连接系统的构建。

用一个简单的示例介绍 Java 中的 gRPC。

```
$ git clone -b v1.19.0 https://github.com/grpc/grpc-java
```

导航到 Java 示例：

```
$ cd grpc-java/examples
```

接下来运行 gRPC 应用程序。首先从 examples 目录编译客户端和服务器：

```
$ ./gradlew installDist
```

运行服务器：

```
$ ./build/install/examples/bin/hello-world-server
```

输出如下信息：

```
三月 04, 2019 2:04:33 下午 io.grpc.examples.helloworld.HelloWorldServer start
```

信息： Server started,listening on 50051

在另一个终端中，运行客户端：

```
$ ./build/install/examples/bin/hello-world-client
```

输出如下信息：

```
三月 04, 2019 2:05:43 下午 io.grpc.examples.helloworld.HelloWorldClient greet
```

信息：Will try to greet world ...

```
三月 04, 2019 2:05:44 下午 io.grpc.examples.helloworld.HelloWorldClient greet
```

信息：Greeting: Hello world

这就是一个使用 gRPC 实现的客户端-服务器应用程序。

现在来看看如何使用服务器上的额外方法更新应用程序以供客户端调用。gRPC 服务是使用 Protobuf 定义的。

现在需要知道的是，服务器和客户端“存根”都有一个 SayHello RPC 方法，该方

法从客户端获取 HelloRequest 参数并从服务器返回 HelloReply，并且此方法定义如下：

```
//问候服务定义
service Greeter {
  //发送问候语
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

//包含用户名的请求消息
message HelloRequest {
  string name = 1;
}

//包含问候语的响应消息
message HelloReply {
  string message = 1;
}
```

让我们更新一下，让 Greeter 服务有两个方法。编辑 src/main/proto/helloworld.proto 并使用新的 SayHelloAgain 方法更新它，用相同的请求和响应类型：

```
//问候语服务定义
service Greeter {
  //发送问候语
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  //发送另一个问候语
  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}
}

//包含用户名的请求消息
message HelloRequest {
  string name = 1;
}

//包含问候语的响应消息
message HelloReply {
  string message = 1;
}
```

当重新编译该示例时，正常编译将重新生成 GreeterGrpc.java，其中包含我们生成的 gRPC 客户端和服务器类。这还会重新生成用于填充、序列化和检索请求和响应类型的类。

但是，我们仍然需要在示例应用程序的人工编写部分实现并调用新方法。

在同一目录中，打开 src/main/java/io/grpc/examples/helloworld/HelloWorldServer.java。像这样实现新方法：

```
private class GreeterImpl extends GreeterGrpc.GreeterImplBase {

  @Override
```

```
    public void sayHello(HelloRequest req, StreamObserver<HelloReply>
responseObserver) {
      HelloReply reply = HelloReply.newBuilder().setMessage("Hello " + req.
getName()).build();
      responseObserver.onNext(reply);
      responseObserver.onCompleted();
    }

    @Override
    public void sayHelloAgain(HelloRequest req, StreamObserver<HelloReply>
responseObserver) {
  HelloReply reply =
       HelloReply.newBuilder().setMessage("Hello again " + req.getName()).build();
      responseObserver.onNext(reply);
      responseObserver.onCompleted();
    }
  }
  ...
```

在同一目录中，打开 src/main/java/io/grpc/examples/helloworld/HelloWorldClient.java。像这样调用新方法：

```
  public void greet(String name) {
    logger.info("Will try to greet " + name + " ...");
    HelloRequest request = HelloRequest.newBuilder().setName(name).build();
    HelloReply response;
    try {
      response = blockingStub.sayHello(request);
    } catch (StatusRuntimeException e) {
      logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
      return;
    }
    logger.info("Greeting: " + response.getMessage());
    try {
      response = blockingStub.sayHelloAgain(request);
    } catch (StatusRuntimeException e) {
      logger.log(Level.WARNING, "RPC failed: {0}", e.getStatus());
      return;
    }
    logger.info("Greeting: " + response.getMessage());
  }
```

就像之前所做的那样，从 examples 目录编译客户端和服务器：

```
  $ ./gradlew installDist
```

运行服务器：

```
  $ ./build/install/examples/bin/hello-world-server
```

在另一个终端中，运行客户端：

```
  $ ./build/install/examples/bin/hello-world-client
```

在 gRPC 中，客户端应用程序可以直接调用不同计算机上的服务器应用程序中的方

法，就像它是本地对象一样，使用户可以更轻松地创建分布式应用程序和服务。与许多 RPC 系统一样，gRPC 基于定义服务的思想，指定可以使用其参数和返回类型远程调用的方法。在服务器端，服务器实现此接口并运行 gRPC 服务器来处理客户端调用。在客户端，客户端有一个存根（在某些语言中称为客户端），它提供与服务器相同的方法。

5.3 本章小结

除了 Raft 协议，还有 Paxos 协议可以用来实现分布式共识。

第 6 章 智能搜索案例分析

本章介绍医药垂直搜索引擎和电商搜索两个应用案例。

6.1 医药垂直搜索引擎

临床实验是一项医学研究。新药/新疗法可以通过临床实验来确定其疗效、安全性及副作用。

这里抓取临床实验信息并通过搜索的形式展示。使用 Elasticsearch 搜索公开的药物临床试验项目信息。首先使用网络爬虫抓取网页和 Word 文档中的信息，然后索引和搜索数据。

采用多项目的方式建立 Gradle 构建。

```
-->Starnderd Location
       -->Project1
          -->settings.gradle
          -->build.gradle
       -->Project2
          -->settings.gradle
          -->build.gradle
       -->build.gradle
       -->settings.gradle
```

3 个项目：爬虫、Web、搜索。

```
-->Med
       -->Crawler
```

```
          -->settings.gradle
          -->build.gradle
        -->Web
          -->settings.gradle
          -->build.gradle
        -->Search
          -->settings.gradle
          -->build.gradle
        -->build.gradle
        -->settings.gradle
```

Gradle 中的多项目构建由一个根项目和一个或多个子项目组成，这些子项目也可能包含子项目。

```
rootProject.name = 'Med'
include 'Search', 'Crawler', 'Web'
```

6.1.1　网络爬虫

爬虫子项目 Crawler 的 build.gradle 文件内容如下：

```
apply plugin:'java'

repositories {
   maven {url 'http://maven.aliyun.com/nexus/content/groups/public/'}
   mavenCentral()
}

dependencies {
   compile group: 'com.squareup.okhttp3', name: 'okhttp', version: '3.13.1'

   testCompile group: 'junit', name: 'junit', version: '4.11'
}
```

然后新建 src/main/java 和 src/test/java 两个源代码目录，分别存放主代码和测试代码。构建 Gradle 包装器：

```
>gradle wrapper
```

主代码中，表示药物临床试验信息的 POJO 类 Clinicaltrials 代码如下：

```
public class Clinicaltrials {
   public String org_study_id;           //研究机构编码
   public String nct_id;                 //项目的 NCT 编号
   public String brief_title;            //简要标题
   public String cn_brief_title;         //中文简要标题
   public String acronym;                //首字母缩略词
   public String official_title;         //正式标题
   public String cn_official_title;      //中文正式标题

   public String lead_sponsor_agency;    //主要赞助机构
```

```
        public String cn_lead_sponsor_agency;          //主要赞助机构中文名

        public String lead_sponsor_agency_class;       //主要赞助商代理类别
        public String cn_lead_sponsor_agency_class;    //主要赞助商代理类别中文描述

        //合作机构
        public ArrayList<String> collaborator_agency = new ArrayList<String>();
        //合作机构中文描述
        public ArrayList<String> cn_collaborator_agency = new ArrayList<String>();
        //合作机构代理类别
        public ArrayList<String> collaborator_agency_class = new ArrayList<String>();
        //合作机构代理类别中文描述
        public ArrayList<String> cn_collaborator_agency_class = new ArrayList
<String>();
        //简要摘要
        public String brief_summary;
        //简要摘要中文翻译
        public String cn_brief_summary;

        public String source;          //来源
        public String cn_source;       //来源中文翻译
        public String has_dmc;         //数据监测委员会 Data Monitoring Committee
        public String cn_has_dmc;
        public String is_fda_regulated_drug;
        public String cn_is_fda_regulated_drug;
        public String is_fda_regulated_device;
        public String cn_is_fda_regulated_device;

        public String detailed_description;    //详细描述
        public String cn_detailed_description;

        public String overall_status;          //项目状态
        public String cn_overall_status;

        public String why_stopped;             //停止原因

        public String start_date;              //开始日期
        public String cn_start_date;

        public String completion_date;         //结束日期
        public String cn_completion_date;

        public String primary_completion_date;
        public String cn_primary_completion_date;

        public String phase;                   //试验阶段
        public String cn_phase;
```

```
        public String study_type;              //研究类型
        public String cn study type;

        public String has expanded access;
        public String cn has expanded access;

        public String allocation;              //分配
        public String cn allocation;
        public String intervention_model;
        public String cn intervention model;
        public String intervention model description;
        public String cn intervention model description;

        public String primary_purpose;         //主要目的
        public String cn primary purpose;

        public String masking;                 //盲法
        public String cn masking;

        public String masking_description;     //盲法描述
        public String cn masking description;

        //主要输出指标
        public ArrayList<String> primary outcome measure = new ArrayList<String>();
        public ArrayList<String> cn_primary_outcome_measure = new ArrayList<String>();
        //主要输出时间窗
        public ArrayList<String> primary outcome time frame = new ArrayList<String>();
        public ArrayList<String> cn_primary_outcome_time_frame = new ArrayList
<String>();
        public ArrayList<String> primary outcome description = new ArrayList
<String>();
        public ArrayList<String> cn primary outcome description = new ArrayLis
t<String>();

        public ArrayList<String> secondary outcome measure = new ArrayList<String>();
        public ArrayList<String> cn secondary outcome measure = new ArrayList
<String>();

        public ArrayList<String> secondary outcome time frame = new ArrayList
<String>();
        public ArrayList<String> cn_secondary_outcome_time_frame = new ArrayList
<String>();

        public ArrayList<String> secondary outcome description = new ArrayList
<String>();
        public ArrayList<String> cn secondary outcome description = new ArrayList
<String>();

        public String enrollment;   //登记
```

```
        public ArrayList<String> conditions = new ArrayList<String>(); //适应症
        public ArrayList<String> cn_conditions = new ArrayList<String>(); //适应
症中文翻译

        public ArrayList<String> arm group arm group label = new ArrayList<String>();
        public ArrayList<String> cn arm group arm group label = new ArrayList
<String>();
        public ArrayList<String> arm group type = new ArrayList<String>();
        public ArrayList<String> cn_arm_group_type = new ArrayList<String>();

        public ArrayList<String> arm group description = new ArrayList<String>();
        public ArrayList<String> cn arm group description = new ArrayList<String>();

        public ArrayList<String> intervention_type = new ArrayList<String>(); //
干预类型
        public ArrayList<String> cn intervention type = new ArrayList<String>();
        public ArrayList<String> intervention_name = new ArrayList<String>();
        public ArrayList<String> cn intervention name = new ArrayList<String>();
        public ArrayList<String> intervention description = new ArrayList<String>();
        public ArrayList<String> cn intervention description = new ArrayList<String>();

        public ArrayList<String> intervention_arm_group_label = new ArrayList
<String>();
        public ArrayList<String> cn intervention arm group label = new ArrayList
<String>();

        public ArrayList<String> intervention_other_name = new ArrayList<String>();
        public ArrayList<String> cn intervention other name = new ArrayList<String>();

        public String criteria;        //招募条件
        public String cn_criteria;     //招募条件中文翻译
        public String gender;          //性别
        public String cn_gender;       //性别中文翻译
        public String minimum_age;     //最小年龄
        public String cn_minimum_age;  //最小年龄中文翻译

        public String maximum_age;     //最大年龄
        public String cn_maximum_age;  //最大年龄中文翻译

        public ArrayList<String> overall_official_last_name = new ArrayList
<String>();                            //姓氏
        public ArrayList<String> overall official role = new ArrayList<String>();
                                       //角色
                                       //角色中文翻译
        public ArrayList<String> cn overall official role = new ArrayList<String>();

        //联系方式
        public ArrayList<String> overall_official_affiliation = new ArrayList
```

```
<String>();
        public ArrayList<String> cn overall official affiliation = new ArrayList
<String>();

        public String overall_contact_last_name;        //联系人姓氏
        public String overall_contact_phone;            //联系电话
        public String overall_contact_phone_ext;        //联系电话分机
        public String overall_contact_email;            //联系邮件

        //位置设施名称
        public ArrayList<String> location facility name = new ArrayList<String>();
        //位置设施名称中文翻译
        public ArrayList<String> cn_location_facility_name = new ArrayList<String>();
        //位置设施地址城市
        public ArrayList<String> location_facility_address_city = new ArrayList
<String>();
        public ArrayList<String> cn location facility address city = new ArrayList
<String>();

        //设施所处州
        public ArrayList<String> location_facility_address_state = new ArrayList
<String>();
        public ArrayList<String> cn location facility address state = new ArrayList
<String>();
        //地址的邮编
        public ArrayList<String> location_facility_address_zip = new ArrayList
<String>();
        //设施所处国家
        public ArrayList<String> location_facility_address_country = new ArrayList
<String>();
        public  ArrayList<String>  cn location facility address country  =  new
ArrayList<String>();

        public ArrayList<String> location_countries = new ArrayList<String>();
//国家
        public ArrayList<String> cn location countries = new ArrayList<String>();

        public String verification_date;            //验证日期
        public String cn_verification_date;         //验证日期中文

        public String study_first_submitted;        //研究首次提交
        public String cn study first submitted;

        public String study_first_submitted_qc;     //研究首次提交质量控制
        public String cn study first submitted qc;

        public String study_first_posted;           //研究首次发表
        public String cn_study_first_posted;
```

```
    public String last_update_submitted;      //上次更新提交时间
    public String cn_last_update_submitted;

    public String last_update_submitted_qc;   //上次更新提交质量控制
    public String cn_last_update_submitted_qc; //上次更新提交质量控制中文翻译

    public String last_update_posted;         //上次更新发布时间
    public String cn_last_update_posted;      //上次更新发布时间中文翻译

    public String keyword;                    //关键词
    public String cn_keyword;

    public String responsible_party_type;     //责任方类型
    public String cn_responsible_party_type;

    public String investigator_affiliation;   //调查员隶属关系
    public String cn_investigator_affiliation;

    public String investigator_full_name;     //调查员全名
    public String investigator_title;         //调查员头衔
    public String cn_investigator_title;      //调查员头衔中文翻译

    public ArrayList<String> mesh_term = new ArrayList<String>();  //网格术语
    public ArrayList<String> cn_mesh_term = new ArrayList<String>();

    public String sharing_ipd;                //是否分享 IPD
    public String cn_sharing_ipd;
    public String ipd_description;            //IPD 描述
    public String cn_ipd_description;         //IPD 描述中文翻译
}
```

在网络连接不稳定的时候，爬虫可以支持放弃下载之前多次重试。最简单的重试代码如下：

```
int retries = 3;
while(true) {
  try {
   downLoad();
   break; //成功!
  } catch(IOException ex) {
   if(--retries == 0) throw new Exception("fail");
   else Thread.sleep(1000);
  }
}
```

Spring Retry 是一个用于实现重试的库，是 Spring 系列的一部分。因为项目是由 Gradle 构建的，所以在 build.gradle 中添加 spring-retry 依赖项：

```
    dependencies {
        ...
        compile group: 'org.springframework', name: 'spring-aspects', version:
'5.0.7.RELEASE'
        compile  group:  'org.springframework.retry',  name:  'spring-retry',
version: '1.2.2.RELEASE'
        ...
    }
```

刷新配置后，Gradle 会下载适当的库。通过添加@Retryable 注释来告诉 Spring 重试指定的方法。默认情况下，在报告失败之前最多进行 3 次尝试。如果要修改尝试次数，则可以在@Retryable 注解中通过 maxAttempts 参数指定尝试次数。

支持重试的下载网页代码如下：

```
    @Component
    public class DownService {
       public static int i = 0;

       @Retryable(maxAttempts = 5)  //最多重试 5 次
       public  Response  getContent(OkHttpClient  client,  String  url)  throws
Exception {
          System.out.println("GetContent Attempt:" + i++);

          okhttp3.Request.Builder rb = new Request.Builder().url(url);
          Request request = rb.build();
          Response response = client.newCall(request).execute();

          int responseCode = response.code();

          String contentType = response.header("Content-Type");
          if (responseCode != 200 || contentType == null) {
             throw new IOException("responseCode " + responseCode);
          }
          return response;
       }

       @Retryable( value = {Exception.class},maxAttempts = 5)  //最多重试 5 次
       public Document getDoc(String url, OkHttpClient client) throws Exception {
          System.out.println("GetDoc Attempt:" + i++);
          okhttp3.Request.Builder rb = new Request.Builder().url(url);
          Request request = rb.build();
          Response response = client.newCall(request).execute();

          int responseCode = response.code();

          String contentType = response.header("Content-Type");
          if (responseCode != 200 || contentType == null) {
             throw new IOException("responseCode " + responseCode);
          }
          String xmlContent = response.body().string();
          Document document = Jsoup.parse(xmlContent, "", Parser.xmlParser());
```

```
            return document;
        }

    }
```

使用这个下载服务得到解析后的文档对象：

```
    OkHttpClient  client  =  new  OkHttpClient.Builder().connectTimeout(1000,
TimeUnit.SECONDS)
                .readTimeout(200, TimeUnit.SECONDS)
                .retryOnConnectionFailure(true).build();

    ConfigurableApplicationContext context = new AnnotationConfigApplicationContext(
                DownFieldCrawlerApplication.class);

    Document document = context.getBean(DownService.class).getDoc(url, client);
```

有些字段是日期类型，例如临床试验的开始日期和结束日期。可以使用模板引擎把这样的英文日期翻译成中文。把匹配英文日期的规则加入文法库的代码如下：

```
    QuestionGrammar g = new QuestionGrammar();    //文法库

    //匹配英文日期的规则
    String right = "<Begin><nt>{word} <num>{daynum}, <num>{yearnum}<End>";
    String handlerName = "Date";                      //处理器名
    g.add(handlerName, right);                        //把处理器加入翻译文法中
```

根据从英文日期文本中提取的年、月、日信息得到翻译结果的处理器：

```
    public class DateHandler implements QuestionHandler {

        @Override
        public String getAnswer(KnowlegeBase kb,Evidence args) {
            //得到英文单词
            String enMon = args.args.get("word");
            String ans = args.args.get("yearnum")+"年"+mon
                    +args.args.get("daynum")+"日";    //生成中文日期
            return ans;
        }

        public static String getBySQL(String input) throws SQLException {
            Connection con = SqliteStore.getConnect();
            QueryRunner runner = new QueryRunner();
            ScalarHandler<String> h = new ScalarHandler<String>();

            String ans = runner.query(con, "select cn from words where word = ?",
                    h, input);
            return ans;
        }

    }
```

使用规则翻译日期的代码如下。

```
KnowlegeBase kb = new KBSqlite();              //使用 Sqlite 存储知识库
GrammarTranslator translator = new GrammarTranslator(kb);

String enDate = "August 24, 2017";             //英文日期
String ans = translator.getResult(enDate);    //中文日期

System.out.println(ans);
```

输出：

```
2017 年 8 月 24 日
```

可以使用 JDBI(https://github.com/jdbi/jdbi)把提取出来的结果存入数据库表。

```
Handle handle = Jdbi.open(getConnect());
Update update = handle.createUpdate(
    "INSERT INTO disease (en) VALUES (:enName)").bind("enName", "1111");

int rows = update.execute();
System.out.println(rows);
```

6.1.2 抓取 PubMed

PubMed 是美国国家医学图书馆（NLM）下属的国家生物技术信息中心（NCBI）开发的医药论文数据库。这个数据库提供了编程接口返回 XML 格式的论文描述信息。PubMedCentral（PMC）是 NLM 的生物医学和生命科学期刊文献的免费全文档案。

可以通过 http://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi 接口按查询关键词抓取论文信息。

首先取得结果数量：

```
public static int getResultNum(CloseableHttpClient client, String term)
        throws Exception {
    String data = new URI(null, term, null).toASCIIString();
    String url = "http://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?
term="
            + data + "&retmax=10&retstart=1";

    String content = HttpUtil.getContent(client, url);

    Document doc = Jsoup.parse(content);
    Element e = doc.getElementsByTag("Count").first();
    String countStr = e.text();

    return Integer.parseInt(countStr);
}
```

使用默认的 HttpClient 对象会报 Invalid cookie header 的警告。解决 Cookie rejected 警告最简单的办法是设置一个空 Cookie。参考代码如下：

```
    //采用用户自定义 cookie 策略，只是使 cookie rejected 的报错不出现
    CookieSpecProvider easySpecProvider = new CookieSpecProvider() {
       @Override
       public CookieSpec create(org.apache.http.protocol.HttpContext arg0) {
          return null;
       }
    };

    Registry<CookieSpecProvider> registry =
     RegistryBuilder.<CookieSpecProvider>create()
              .register("easy", easySpecProvider)
              .build();

    CloseableHttpClient client = HttpClients.custom()
           .setDefaultCookieSpecRegistry(registry)
           .build();
```

然后根据结果数量设定要抓取多少条结果：

```
    public static void insertIDS(CloseableHttpClient client, String term) throws
Exception{
       Connection con = DBUtil.getConnect();
       int resultNum = getResultNum(client, term);

       String data = new URI(null, term, null).toASCIIString();
       String url ="http://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?
term="
            + data + "&retmax="+resultNum+"&retstart=1";
       String content = HttpUtil.getContent(client, url);
       Document doc = Jsoup.parse(content);
       Elements tables = doc.getElementsByTag("Id");

       String sql = "insert into paper(id)values(?)";
       PreparedStatement insertStmt = con.prepareStatement(sql);

       for (Element e : tables) {
          String id = e.text();
          insertStmt.setString(1, id);
          insertStmt.executeUpdate();
       }
       con.commit();
    }
```

把抓取下来的论文信息存入 PostgreSQL 数据库。

```
    Class.forName("org.postgresql.Driver");
    String jdbcUrl = "jdbc:postgresql://localhost:5432/postgres";
    String username = "postgres";
    String password = "mysecretpassword";

    try (Connection conn = DriverManager.getConnection(jdbcUrl, username, password);
       String sql = "insert into paper(id)values(?)";
       PreparedStatement insertStmt = conn.prepareStatement(sql);
```

```
        for (Element e : tables) {
           String id = e.text();
           insertStmt.setString(1, id);
           insertStmt.executeUpdate();
        }

     } catch (SQLException e) {
        e.printStackTrace();
     }
```

6.1.3　MVC 搜索界面开发

实现搜索展示的 Freemarker 模板的 search-nextpage.ftl 内容如下：

```
    <html>
    <head>
       <title>${websiteTitle}</title>
       <META http-equiv=Content-Type content="text/html; charset=utf-8">
       <link href="CSS/default.css" rel="stylesheet" type="text/css" />
    </head>
    <body>
       <div id="sHeader">
          <div id="searcher">
            <form method="get" action="search">
                <input  ID="tbKeyword"  type="text"  name="query"  class="input
text" value="${query}" />
                <input ID="btSearch" type=submit value="试着搜搜"/>
            </form>
          </div>
          </div>
       <div id="content">

          <div id="searcherContent">
             <div id="main">

                 <div id="search-report">约有记录：${totalsize}条</div>

    <#list resultItems as result>
                        <div class="search-item">
                           <b><a href='${result.url}' target="_blank">${result.
title}</a></b>
                        </div>

                        <div class="search-item">
                           ${result.body}
                        </div>
    </#list>
       <#if (pager.hasNextPage())>
          <a  href="${pager.getNextUrl()}"  class="rnavLink">下一页 &#187;
</a>
```

```
        </#if>
                </div>
            </div>

        </div>
    </body>
    </html>
```

后台控制类通过 BeanFactory 得到 RestHighLevelClient 的实例：

```
    @RestController
    @SpringBootApplication
    public class SearchESNextURL extends SpringBootServletInitializer {
        //从配置文件 application.properties 读取 es 服务主机名
        @Value("${elasticsearch.host}")
        private String host;

        //从配置文件 application.properties 读取 es 服务监听端口号
        @Value("${elasticsearch.port}")
        private int port;

        @Bean(name="searchService")
        public RestHighLevelClient restHighLevelClient() {
           return
          new  RestHighLevelClient(RestClient.builder(new  HttpHost(host,  port,
"http")));
        }

        @Autowired
        private BeanFactory beanFactory;
    }
```

得到 RestHighLevelClient 对象的代码如下：

```
    RestHighLevelClient client =
             beanFactory.getBean("searchService",RestHighLevelClient.class);
```

SearchESNextURL 控制类的 search()方法实现如下：

```
    @RequestMapping("/search")
    public ModelAndView search(@RequestParam(value = "query") String q,
          @RequestParam(required = false, value = "pager.offset") Integer offset)
          throws TemplateException, IOException {
       ModelAndView mv = new ModelAndView("search-nextpage");
       mv.getModelMap().addAttribute("websiteTitle", "search Demo " + q);

       fillModel(mv.getModelMap(), q, offset);
       return mv;
    }
```

fillModel()方法实现如下：

```
    public void fillModel(Map<String, Object> modelMap,
                   String keyWords, Integer offset) throws IOException {
       List<WebItem> items = new ArrayList<>();
```

```
String titleField = "title";
String bodyField = "body";

MatchPhraseQueryBuilder pqBody =
       QueryBuilders.matchPhraseQuery(bodyField, keyWords);

MatchPhraseQueryBuilder pqTitle =
      QueryBuilders.matchPhraseQuery(titleField, keyWords);

QueryStringQueryBuilder fuzzyQb = new QueryStringQueryBuilder(keyWords);

QueryBuilder qb =
  QueryBuilders.boolQuery().should(pqBody).should(pqTitle).should(fuzzyQb);

String index = "news";                            //索引名

SearchRequest request = new SearchRequest(index);

SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
sourceBuilder.query(qb);

int size = 20;
if (offset != null) {
   sourceBuilder.from(offset.intValue());
}

//搜索结果返回条数默认值为 10
sourceBuilder.size(size);

request.source(sourceBuilder);

HighlightBuilder highlightBuilder = new HighlightBuilder();
HighlightBuilder.Field highlightTitle =
      new HighlightBuilder.Field(titleField);       //title 字段高亮

highlightTitle.highlighterType("unified");         //配置高亮类型

highlightBuilder.field(highlightTitle);            //添加到 builder
HighlightBuilder.Field highlightBody = new HighlightBuilder.Field(bodyField);
highlightBuilder.field(highlightBody);

highlightBuilder.preTags("<span style=\"color:red\">");
highlightBuilder.postTags("</span>");

sourceBuilder.highlighter(highlightBuilder);

RestHighLevelClient client =
    beanFactory.getBean(RestHighLevelClient.class, "searchService");

SearchResponse searchResponse =
```

```
            client.search(request, RequestOptions.DEFAULT);

        SearchHits hits = searchResponse.getHits();

        long totalHits = hits.getTotalHits();          //得到结果总数
        PagerTag pagerTag = new PagerTag("./search", totalHits);

        if (offset != null) {
            pagerTag.setOffset(offset.intValue());
        }
        pagerTag.setMaxPageItems(size);
        pagerTag.addParam("query", keyWords);

        modelMap.put("pager", pagerTag);

        modelMap.put("query", keyWords);
        modelMap.put("totalsize", String.valueOf(totalHits));

        for (SearchHit hit : hits) {
            Map<String, Object> result = hit.getSourceAsMap();
            WebItem webItem = new WebItem((String) result.get("url"),
                     (String) result.get("title"),
                    (String) result.get("body"));

            HighlightField titleHighlight = hit.getHighlightFields().get(titleField);

            if (titleHighlight != null) {
                Text[] text = titleHighlight.fragments();

                String fragmentString = StringUtils.join(text, "...");
                webItem.setTitle(fragmentString);
            }

            HighlightField bodyHighlight = hit.getHighlightFields().get(bodyField);

            if (bodyHighlight != null) {
                Text[] text = bodyHighlight.fragments();

                String fragmentString = StringUtils.join(text, "...");
                webItem.setBody(fragmentString);
            }

            items.add(webItem);
        }

        modelMap.put("resultItems", items);
    }
```

6.1.4　构建知识库

可以从结构化文本中构建知识图谱。

三元组：<医院治疗疾病>

例如，从 https://clinicaltrials.gov/show/NCT02905136?resultsxml=true 提取出：

```
< Hospices Civils de Lyon , cure , Autoimmune encephalitis >
```

首先调用翻译 API 把“自身免疫性脑炎”翻译成“Autoimmune encephalitis”。代码如下：

```
String text = "自身免疫性脑炎";

OkHttpClient client =
    new OkHttpClient.Builder().connectTimeout(1000, TimeUnit.SECONDS)
      .readTimeout(200,  TimeUnit.SECONDS).retryOnConnectionFailure(true).
build();
Translate.setKey(ApiKeys.YANDEX_API_KEY);
Language source = Language.CHINESE;
Language target = Language.ENGLISH;

String translation = Translate.execute(client, text, source, target);
System.out.println("Translation: " + translation);
```

可以使用 roslyn(https://github.com/dotnet/roslyn)生成表示相关知识的代码。

首先安装包：

```
PM> Install-Package Microsoft.CodeAnalysis
PM> Install-Package Microsoft.CSharp
PM> Install-Package Microsoft.CodeAnalysis.CSharp
```

使用如下代码生成适应症（Disorder）类：

```
//创建命名空间：(命名空间 CodeGenerationSample)
var @namespace =
     SyntaxFactory.NamespaceDeclaration(
     SyntaxFactory.ParseName("CodeGenerationSample")).NormalizeWhitespace();

//添加系统使用语句：(using System)
@namespace =
  @namespace.AddUsings(
     SyntaxFactory.UsingDirective(SyntaxFactory.ParseName("System")));

//创建一个类：(Disorder 类)
var classDeclaration = SyntaxFactory.ClassDeclaration("Disorder");

//添加 public 修饰符：(public class Disorder)
classDeclaration =
    classDeclaration.AddModifiers(SyntaxFactory.Token(SyntaxKind.PublicKeyword));

//创建一个属性：(public string DisorderName { get; set; })
```

```
    var propertyDeclaration =
        SyntaxFactory.PropertyDeclaration(SyntaxFactory.ParseTypeName("string"),
"DisorderName")
       .AddModifiers(SyntaxFactory.Token(SyntaxKind.PublicKeyword))
       .AddAccessorListAccessors(

       SyntaxFactory.AccessorDeclaration(SyntaxKind.GetAccessorDeclaration).
        WithSemicolonToken(SyntaxFactory.Token(SyntaxKind.SemicolonToken)),

       SyntaxFactory.AccessorDeclaration(SyntaxKind.SetAccessorDeclaration).
        WithSemicolonToken(SyntaxFactory.Token(SyntaxKind.SemicolonToken)));

    //将字段、属性和方法添加到类中
    classDeclaration = classDeclaration.AddMembers( propertyDeclaration);

    //将类添加到命名空间
    @namespace = @namespace.AddMembers(classDeclaration);

    //规范化并得到字符串形式的代码
    var code = @namespace
       .NormalizeWhitespace()
       .ToFullString();

    //将新代码输出到控制台
    Console.WriteLine(code);
```

这里要注意的最重要的事情是所有 Roslyn 对象都是不可变的。所以这样写：

```
    @namespace.AddUsings(
         SyntaxFactory.UsingDirective(SyntaxFactory.ParseName("System")));
```

AddUsings 不会改变 namespace 对象。它返回一个带有更改的新的 CompilationUnitSyntax。所以必须这样写：

```
    @namespace =
      @namespace.AddUsings(
         SyntaxFactory.UsingDirective(SyntaxFactory.ParseName("System")));
```

使用 roslyn 解析代码。

在前期，这样的三元组数据可以直接存入 Sqlite 这样的数据库；在后期，可以把数据导入 Elasticsearch 这样的文档型存储库。

为了迁移知识库，可以使用 elasticdump(https://github.com/taskrabbit/elasticsearch-dump)导出数据。

安装：

```
    #npm install elasticdump -g
```

使用：

```
    #elasticdump \
      --input=http://47.93.206.182:9200/news \
```

```
    --output=/data/my index.json \
    --type=data
```

6.1.5　自动问答

在搜索页面可以集成自动问答。不同意图的问句需要不同的处理方式。例如“糖尿病如何治疗”和“如何预防癌症”需要不同的处理器。

有很多个处理器同时匹配一个问句。现有的一些处理器包括：处理简单问答对用的 ChatHandler，处理治疗疾病的 CureHandler，处理英文单词的 WordHandler，等。

处理器从问句提取关键词，提取出来的关键词以键/值对的形式存入 PairListString 对象：

```
//键可以重复
public class PairListString {
   String[] values;  //存储所有名称和值
   int count;

   public PairListString(int initialCapacity) {
      values = new String[initialCapacity * 2];
   }

   /**
    * 增加一个名称/值对
    *
    * @param x 名称
    * @param y 名称对应的值
    */
   public void addPair(String x, String y) {
      if (count * 2 >= values.length) {
         values = Arrays.copyOf(values, values.length * 2);
      }
      values[count * 2] = x;
      values[count * 2 + 1] = y;
      count++;
   }

   public String getX(int index) {  //根据下标得到键
      return values[index * 2];
   }

   public String getY(int index) {  //根据下标得到值
      return values[index * 2 + 1];
   }

   public String get(String key){  //得到键对应的值
```

```
        for (int i = 0; i < count; ++i) {
          if(values[i * 2].equals(key)){
              return values[i * 2 + 1];
          }
        }

        return null;
    }
}
```

使用这个类存放从问句“糖尿病怎么治疗”中提取的参数，示例代码如下：

```
PairListString questionArgs = new PairListString(1);

String type = "DiseaseName";                    //键
String diseaseName = "糖尿病";                   //值

questionArgs.addPair(type, diseaseName);
System.out.println(questionArgs.get(type));    //输出 糖尿病
```

动态加载问句处理器。例如：

```
String handleClass = "questionHandler."+handleName+"Handler";//得到类名
Class<? extends QuestionHandler> clz = Class.forName(handleClass)
       .asSubclass(QuestionHandler.class);    //返回 QuestionHandler 的子类
QuestionHandler answer = clz.newInstance();    //得到问句处理器
String ans = answer.getAnswer(kb,g.questionArgs); //依据问句处理器返回答案
```

一个问句所得到的规则对象。

```
public class Rule {
   public ArrayList<String> rhs =
               new ArrayList<String>();        //右边的 Token 类型序列
   public ArrayList<TokenType> lhs =
               new ArrayList<TokenType>();     //左边的 Token 类型序列
   public HashMap<String, HashSet<String>> words =
              new HashMap<String, HashSet<String>>();//词表
}
```

处理问句的规则：

```
String ruleStr = "<num>{plusnum1}加<num>{plusnum2}";
ArrayList<RuleToken> tokens = IERuleParser.getSeq(ruleStr,67); //规则以及编号
for (RuleToken t : tokens) {
   System.out.println(t);                      //输出规则中的每个 Token
}
Rule rule = RuleBuilder.create(tokens);        //根据 Token 序列创建规则
System.out.println(rule);                      //输出生成的问句规则
System.out.println("is valid:"+rule.valid());//验证问句规则是否有效
```

根据问句模板处理问句：

```
QuestionGrammar g = new QuestionGrammar();     //问句文法库
```

```
String right = "<Begin><num>{plusnum1}加<num>{plusnum2}<End>";  //问句模板
g.add("Add", right);                                  //问句意图:Add

right = "<Begin><DiseaseName>{disease}怎么治疗<End>";
g.add("Cure", right);                                 //问句意图:Cure

String type = "DiseaseName";
String diseaseName = "糖尿病";
//仅仅增加词，而不是加规则
g.addWord(diseaseName, type);

TextExtractor ie = new TextExtractor(g);              //问句文法对应的文本提取器

String question = "糖尿病怎么治疗";
AdjList adjList = ie.getLattice(question);            //返回问句对应的意图
System.out.println(adjList);
```

可以使用 Sqlite 数据库或者使用 Elasticsearch 存储知识库，因此将知识库定义成为一个接口：

```
public interface KnowlegeBase {
   public String getCureMethod(String disease) throws Exception;
                                                      //疾病的处置方法
   public String getChat(String inputQuestion) throws Exception;//简单问答对
}
```

Sqlite 数据库存储知识的实现如下：

```
public class KBSqlite implements KnowlegeBase{
   static Connection con = SqliteStore.getConnect();  //数据库连接

   @Override
   public String getCureMethod(String disease) throws Exception {
      QueryRunner runner = new QueryRunner();
      ScalarHandler<String> h = new ScalarHandler<String>();
      //查询治疗方法表返回答案
      String ans = runner.query(con,
            "select answer from cure where disease = ?", h, disease);
      return ans;
   }

   @Override
   public String getChat(String inputQuestion) throws Exception{
      QueryRunner runner = new QueryRunner();
      ScalarHandler<String> h = new ScalarHandler<String>();
      //查询 chat 表返回答案
      String ans = runner.query(con,
            "select answer from chat where question = ?", h, inputQuestion);
      return ans;
```

```
        }
    }
```

根据问句模板返回答案：

```
    KnowlegeBase kb = new KBSqlite();              //使用 Sqlite 数据库存储知识库
    GrammarAnswer answer = new GrammarAnswer(kb);//根据文法返回答案

    String question = "糖尿病怎么治疗";
    String ans = answer.getAnswer(question);       //返回知识库中存储的答案

    System.out.println(ans);                       //输出答案
```

6.2 电商搜索

网上购物时，可能会比较多个网站中的同类商品。可以通过官方商城购物搜索或爬虫抓取所有的官方商城，例如阿迪达斯、小米等，实现多个购物网站的一站式搜索。

6.2.1 电商爬虫

如果需要调整 DNS 服务器地址，则可以考虑用 PowerShell 设置：

```
    Set-DNSClientServerAddress -interfaceIndex 12 -ServerAddresses ("1.2.4.8",
"114.114.114.114")
```

安装所需要的包：

```
    Install-Package Selenium.Support
    Install-Package  Selenium.WebDriver
    Install-Package  Selenium.Firefox.WebDriver
```

如果要使用 Selenium 控制 FireFox，首先可以从 https://github.com/mozilla/geckodriver/releases 下载驱动。例子代码如下：

```
    using OpenQA.Selenium;
    using OpenQA.Selenium.Firefox;

    namespace ConsoleApp1
    {
        public class Program
        {
            public static void Main(string[] args)
            {
                IWebDriver driver = new FirefoxDriver();
                driver.Navigate().GoToUrl("http://www.selenium.academy/Examples/
Interaction.html");
```

```
            IWebElement button = driver.FindElement(By.Id("button"));
            driver.Quit();

            //Wait to exit.
            Console.Read();
        }
    }
}
```

使用 CefSharp(https://github.com/cefsharp/CefSharp)下载动态网页：

```
Install-Package CefSharp.OffScreen
Install-Package CefSharp.Common
Install-Package CefSharp.Wpf
Install-Package CefSharp.WinForms
```

这里抓取 http://www.xiu.com/中的商品，然后导入自己的网上商城。网上商城是用 ECShop 软件搭建的。

将爬虫抓取的信息写入商品表 ecs_goods 和商品类别表 ecs_category。商品表 ecs_goods 的说明如表 6-1 所示。

表 6-1　商品表 ecs_goods

字　段	类　型	说　明
goods_id	mediumint(8)	商品的自增 id
cat_id	smallint(5)	商品所属商品分类 id，取值 ecs_category 的 cat_id
goods_sn	varchar(60)	商品的唯一货号
goods_name	varchar(120)	商品的名称
goods_name_style	varchar(60)	商品名称显示的样式
click_count	int(10)	商品点击数
brand_id	smallint(5)	品牌 id
provider_name	varchar(100)	供应商名称
goods_number	smallint(5)	商品库存数量
goods_weight	decimal(10,3)	商品的重量，以千克为单位
market_price	decimal(10,2)	市场售价
shop_price	decimal(10,2)	本店售价
promote_price	decimal(10,2)	促销价格
promote_start_date	int(11)	促销价格开始日期
promote_end_date	int(11)	促销价格结束日期
warn_number	tinyint(3)	商品报警数量
keywords	varchar(255)	商品关键字，放在商品页的关键字中，为搜索引擎收录用

续表

字　　段	类　　型	说　　明
goods_brief	varchar(255)	商品的简短描述
goods_desc	text	商品的详细描述
goods_thumb	varchar(255)	商品在前台显示的微缩图片，如在分类筛选时显示的小图片
goods_img	varchar(255)	商品的实际大小图片，如进入该商品页时介绍商品属性所显示的大图片
original_img	varchar(255)	商品原图
is_real	tinyint(3)	是否是实物：1—是；0—否；比如虚拟卡就为 0，不是实物
extension_code	varchar(30)	商品的扩展属性，比如像虚拟卡
is_on_sale	tinyint(1)	该商品是否开放销售：1—是；0—否
is_alone_sale	tinyint(1)	是否能单独销售，1—是；0—否；如果不能单独销售，则只能作为某商品的配件或者赠品销售
integral	int(10)	购买该商品可以使用的积分数量
add_time	int(10)	商品的添加时间
sort_order	smallint(4)	商品的显示顺序
is_delete	tinyint(1)	商品是否已经删除，0—否；1—已删除
is_best	tinyint(1)	是否是精品：0—否；1—是
is_new	tinyint(1)	是否是新品：0—否；1—是
is_hot	tinyint(1)	是否热销：0—否；1—是
is_promote	tinyint(1)	是否特价促销：0—否；1—是
bonus_type_id	tinyint(3)	购买该商品所能领到的红包类型
last_update	int(10)	最近一次更新商品配置的时间
goods_type	smallint(5)	商品所属类型 id，取值表 goods_type 的 cat_id
seller_note	varchar(255)	商品的商家备注，仅商家可见
give_integral	int(11)	购买该商品时每笔成功交易赠送的积分数量

记录商品分类信息的商品类别表 ecs_category 部分字段说明如表 6-2 所示。

表 6-2　商品类别表 ecs_category 部分字段说明

字　　段	类　　型	说　　明
cat_id	smallint(5)	自增 id 号
cat_name	varchar(90)	类别名称
keywords	varchar(255)	类别的关键字描述
cat_desc	varchar(255)	类别描述

续表

字 段	类 型	说 明
parent_id	smallint(5)	该类别的父 id，取值于该表的 cat_id 字段
sort_order	tinyint(1)	该类别在页面显示的顺序，数字越大顺序越靠后
template_file	varchar(50)	该类别的模板文件的名字
measure_unit	varchar(15)	该类别的计量单位
show_in_nav	tinyint(1)	是否显示在导航栏，0—不；1—显示在导航栏
style	varchar(150)	该类别单独样式表的包括文件名部分的文件路径
is_show	tinyint(1)	是否在前台页面显示，1—显示；0—不显示
grade	tinyint(4)	该类别的最高和最低价之间的价格分级，当大于 1 时，会根据最大最小价格区间分成区间，会在页面显示价格范围，如 0～300,300～600,600～900

把采集的数据写入 MySQL 数据库。连接 MySQL 数据库的方法是：

```
//设置连接数据库的用户名和密码
Properties props = new Properties();
props.put("user", "root"); //用户名
props.put("password", "password"); //密码

//连接参数，指定网址和数据库名
Connection conn = DriverManager.getConnection(
        "jdbc:mysql://118.145.6.205/s2266", props); //IP 地址/数据库名
```

用 Commons VFS 把下载的商品图片上传到商城服务器。Commons VFS 底层使用 Apache Commons Net 实现 FTP 上传的功能，也可以直接使用 Apache Commons Net 把文件上传到 FTP。

登录到 FTP 服务器：

```
FTPClient ftp = new FTPClient();
ftp.connect("192.168.14.117"); //连接 FTP 服务器
ftp.login("admin", "123"); //登录
```

设置上传文件的类型为二进制：

```
ftp.setFileType(FTP.BINARY_FILE_TYPE);
```

默认上传到根路径，如果想要上传到另外的路径，就改变当前工作路径。改变工作目录到图片所在目录：

```
ftp.changeWorkingDirectory("/admin/pic");
```

检查 FTP 连接是否成功了。

```
int reply = ftp.getReplyCode();
if(FTPReply.isPositiveCompletion(reply)){
  System.out.println("连接成功");
}
```

上传文件到 FTP 服务器。

```
File f1 = new File(location); //本地文件
in = new FileInputStream(f1);
ftp.storeFile("test.jpg",in); //上传后的名字是 test.jpg
```

断开和服务器建立的连接。

```
ftp.logout();
```

6.2.2 商品搜索

搜索“水”，可以显示“饮用水”“化妆水”之类的相关商品类别。按类别统计搜索引擎命中结果，称为聚合搜索（faceted search）。分类可以是多层次的，用户可以沿着某一类继续细化，也可以按类别浏览更多的同类商品。

一旦在 cat 列上启用了 fielddata，就可以使用 textfield 执行与基数聚合关联的查询。例如：

```
curl -XPUT 'localhost:9200/goods/ mapping/personal?pretty' -d'
{
  "properties": {
    "cat": {
      "type":     "text",
      "fielddata": true
    }
  }
}
' -H 'Content-Type: application/json'
{
  "acknowledged" : true
}
```

在类别上执行基数聚合查询：

```
curl -XPOST 'localhost:9200/ goods/ search?&pretty' -d'
{
   "size" : 0,
    "aggs" : {
        "cat count" : {
             "cardinality" : {
                 "field" : "cat"
             }
         }
    }
}
' -H 'Content-Type: application/json'
```

在程序中执行聚合查询：

```
QueryBuilder qb = QueryBuilders.matchAllQuery();
String index = "goods";            //索引名
```

```
SearchSourceBuilder sourceBuilder = new SearchSourceBuilder();
sourceBuilder.query(qb);

TermsAggregationBuilder ab = AggregationBuilders.terms("cat")
      .field("cat");              //分类列
sourceBuilder.aggregation(ab);
```

商品分类树：

```
public class CatNode {
   public int no;                  //类别编号
   public String name;             //类别名称
   public boolean isLeaf;          //是否叶子节点
   public List<CatNode> children = null;      //孩子节点

   public CatNode parent;

   public int level;               //层级

   public CatNode(int no, String name, CatNode parentNo, int l, boolean isLeaf) {
      this.no = no;
      this.name = name;
      this.parent = parentNo;
      this.level = l;
      this.isLeaf = isLeaf;
      if (!isLeaf) {
         children = new ArrayList<CatNode>(5);
      }
   }

   public void addChildren(CatNode node) throws Exception {
      if (isLeaf) {
         throw new Exception("add child error to leaf node:" + no);
      }
      children.add(node);
   }

   public String toString() {
      String temp = this.name;
      for (CatNode child : children) {
         temp += "\n" + "child:" + child.no + ":" + child.name;
      }
      temp += "\n";
      return temp;
   }
}
```

为了提供面包屑导航，需要返回任意指定类别的前 3 级父节点信息。首先定义包含前 3 级父节点信息的 FirstLevels 类：

```
public class FirstLevels {
```

```
    int level1ID;                //第一级节点编号
    int level2ID;                //第二级节点编号
    int level3ID;                //第三级节点编号
    String catName1;             //第一级节点名
    String catName2;             //第二级节点名
    String catName3;             //第三级节点名
}
```

得到给定节点的前 3 级父节点信息：

```
public void getFirstLevels(int id, FirstLevels f) {
    CatNode current = this.get(id);

    if (current.level > 3) {  //如果是第 4 级，则向上取父节点
        current = this.get(current.parent.no);
    }

    if (current.level == 3) { //取得第 3 级父节点
        f.level3ID = current.no;
        f.catName3 = current.name;
        current = this.get(current.parent.no);
    }

    if (current.level == 2) { //取得第 2 级父节点
        f.level2ID = current.no;
        f.catName2 = current.name;
        current = this.get(current.parent.no);
    }

    if (current.level == 1) { //取得第 1 级父节点
        f.level1ID = current.no;
        f.catName1 = current.name;
        current = this.get(current.parent.no);
    }

}
```

记录商品类别信息的节点映射散列表：

```
public class CategoryMap extends HashMap<Integer, CatNode> {
    public String toString() {
        StringBuffer temp = new StringBuffer();

        for (Map.Entry<Integer, CatNode> e : this.entrySet()) {
            temp.append(e.getKey());
            temp.append(':');
            temp.append(e.getValue());
        }

        return temp.toString();
    }
}
```

根据父子关系表生成节点映射类。

```
    String sql = "SELECT ID,isnull(FATHER_ID,0) FATHER_ID,CAT_NAME,CAT_LEVEL FROM
GOODS_CAT order by ID";                              //查询商品类别表
    stmt = con.prepareStatement(sql);

    rs = stmt.executeQuery();

    CatNode thisNode = new CatNode(0, "ROOT", null, 0, false);  //根节点
    this.put(0, thisNode);                           //首先将根节点放入映射表

    while (rs.next()) {
       int code = rs.getInt("ID");                   //节点编号
       String name = rs.getString("CAT_NAME");       //节点名
       int fatherID = rs.getInt("FATHER_ID");        //父节点编号
       int level = rs.getInt("CAT_LEVEL");           //节点级别
       boolean isLeaf = true;                        //是否叶子节点

       CatNode parentNode = this.get(fatherID);      //父节点

       thisNode = new CatNode(code, name, parentNode, level, isLeaf);  //当前节点

       if (parentNode.isLeaf) {
          parentNode.isLeaf = false;                 //修改是否有孩子节点的属性值
          parentNode.children = new ArrayList<CatNode>(5);
       }
       parentNode.children.add(thisNode);            //设置当前节点的父子节点关系
       this.put(code, thisNode);                     //把当前节点放入节点映射表
    }
```

6.2.3 在线客服

浏览公司网站的潜在客户需要在线客服提供一对一的解答和服务。为了节省人力成本，可以开发自动客服，回答一些简单和常见的问题。网站的客服需要根据上下文给出回答。WebSocket 可以实现客户端和服务器端的长连接，记住上下文信息。

WebSocket 是 HTML5 开始提供的一种浏览器与服务器间进行全双工通信的网络技术。使用 WS 或者 WSS 协议的 WebSocket 允许服务器端发起通信，实现双向实时通信。

WebSocket 的工作流程是：浏览器通过 JavaScript 向服务器端发出建立 WebSocket 连接的请求，在 WebSocket 连接建立成功后，客户端和服务器端就可以通过 TCP 连接传输数据。

WebSocket 的服务器端选择用 Java 实现，可以运行在支持 WebScoket 的 Web 服务器中，这里采用 Tomcat 中的 WebScoket 实现。而客户端则是运行在浏览器中包含

JavaScript 调用的 HTML 网页。只有一些新的浏览器（如 Chrome 或者 FireFox）支持 WebScoket。

为了方便测试，可以在嵌入式 Tomcat 中开发 WebSocket 应用。为了使用嵌入式 Tomcat，首先在 build.gradle 中增加如下依赖项：

```
dependencies {
    compile 'org.apache.tomcat.embed:tomcat-embed-core:8.5.20'
}
```

然后在 Eclipse 中创建源代码目录 src/main/java。在 main()方法中启动嵌入式 Tomcat：

```
Tomcat tomcat = new Tomcat();
tomcat.setPort(8080);  //设置监听端口号

//设置服务器的基本目录
File r=new File("");
String basePath=r.getAbsolutePath() + "/src/main/resources/static";
tomcat.setBaseDir(basePath);

//启动服务器
tomcat.start();
//让当前线程等待服务器关闭
tomcat.getServer().await();
```

如下命令运行项目：

```
>gradlew bootRun
```

WebScoket 服务器端代码如下：

```
/**
 * WebSocket 服务器端点实现通过@ServerEndpoint 注解指定客户端访问的 URL 地址
 */
@ServerEndpoint("/Chat/{who}")
public class Chat {
   /**
    * 连接建立成功调用的方法
    * @param session  可选的参数。需要通过 session 给客户端发送数据
    */
   @OnOpen
   public void onMessage(@PathParam("who") String who, Session session) {
      push("欢迎与机器人对话", session);
   }

    /**
     * 收到客户端消息后调用这个方法
     * @param message 客户端发送过来的消息
     * @param session 可选的参数
     */
   @OnMessage
```

```
        public void onMessage(@PathParam("who") String who, String message,
              Session session) {
              String ans = "hi";
              push("机器人回答: " + ans, session);    //向客户端返回信息
        }

        /**
         * 发生错误时调用
         * @param session
         * @param error
         */
        @OnError
        public void onError(Session session,
                  java.lang.Throwable throwable){
           System.out.println("client onError executed." +throwable);
        }

        /**
         * 关闭连接
         */
        @OnClose
        public void onClose() {
            String message = "has disconnection.";
            System.out.println(message);
        }

        /**
         * 发送消息给客户端
         */
        public void push(String message, Session session) {
           session.getAsyncRemote().sendText(message);
        }
    }
```

网页客户端首先引入 reconnecting-websocket.js：

```
    <script  language="JavaScript"  type="text/javascript"  src="reconnecting-
websocket.js"></script>
```

使用 WebSocket 实现和服务器端聊天的网页源代码如下。

```
    <body>
      <div id="logs"></div>        //用于显示对话历史记录
      <div>
    <input id="msg" type="text" placeholder="消息"
             onkeydown = "if (event.keyCode == 13) document.getElementById
('btnSearch').click()"   //处理回车符
                 />
        <button id="btnSearch" type="submit" value="talk" onClick="talk()">说话
</button>
      </div>
```

```
    <script>
      var iam = '我';
      var host = location.origin.replace(/^http/, 'ws');   //主机名
      var context  = window.location.pathname.substring(0, window.location.
pathname.indexOf('/', 2));                                //路径
      var url = host + context + '/Chat/' + iam;          //服务器端的 WS 地址
      var ws = new ReconnectingWebSocket(url);            //和服务器端建立连接

      ws.onmessage = function (message) {                 //处理服务器端返回的消息内容
        var tag = document.createElement('p');
        tag.appendChild(document.createTextNode(message.data));
        document.getElementById('logs').appendChild(tag);
      };

      function talk() {                                   //向服务器端发送消息
        var msg = document.getElementById('msg').value;
        if (msg) {
            ws.send(msg);                                 //通过 WebSocket 发送消息
            document.getElementById('msg').value = '';    //重置输入框中的消息内容
            var tag = document.createElement('p');
            tag.appendChild(document.createTextNode(iam +':'+msg));
            document.getElementById('logs').appendChild(tag);  //记录发送的消息历史
        }
      }
    </script>
  </body>
```

6.3 本章小结

本章首先介绍了集成医药临床信息和论文抓取的医药垂直搜索引擎。在搜索结果页还可以增加知识图谱的展示。

然后在电商搜索小节介绍了电商爬虫和搜索，以及网站在线客服的开发。如果需要搭建电商网站，可以参考 SimplCommerce(https://github.com/simplcommerce/SimplCommerce)，还可以开发手机客户端商品搜索，方便浏览商品描述信息，以及商品评价、相关推荐等。

参 考 文 献

[1] 罗刚，张子宪，崔智杰. Java 中文文本信息处理——从海量到精准[M].北京：清华大学出版社，2017.

[2] 罗刚. 解密搜索引擎技术实战——Lucene & Java 精华版[M]. 3 版. 北京：电子工业出版社，2016.

[3] 罗刚. 网络爬虫全解析——技术、原理与实践[M].北京：电子工业出版社，2017.

[4] 罗刚. 自己动手写网络爬虫(修订版) [M].北京：清华大学出版社，2016.

[5] 罗刚. 使用 C#开发搜索引擎 [M]. 2 版. 北京：清华大学出版社，2018.

这本书浅显易懂，深入浅出，可以帮助读者快速进入搜索引擎研发大门，掌握搜索引擎的科技竞争力。

—— 童小军 红象云腾公司 创始人/CEO

内容翔实，注重实际操作，非常适合喜欢学习并乐于实践的开发者。

——王提楹 京东集团 高级软件工程师

这是一本非常实用的工具书，本书从搭建搜索集群框架基础开始，结合操作案例的演示，全面、翔实地介绍了使用ElasticSearch的具体方法和步骤，引导读者从零开始，一步一步掌握分布式检索的原理和全过程，非常适合做搜索引擎行业的编程人员使用。

——王维艳 泰为信息科技上海公司 测试工程师

课件下载·样书申请



书圈

清华社官方微信号



扫 我 有 惊 喜


ISBN 978-7-302-53550-8
9 787302 535508 >
定价：69.80元